The VLDB Journal
DOI 10.1007/s00778-012-0278-6

REGULAR PAPER

Scalable and dynamically balanced shared-everything OLTP with physiological partitioning

Pınar Tözün · Ippokratis Pandis · Ryan Johnson · Anastasia Ailamaki

Received: 7 September 2011 / Revised: 21 January 2012 / Accepted: 2 May 2012

Abstract Scaling the performance of shared-everything transaction processing systems to highly parallel multicore hardware remains a challenge for database system designers. Recent proposals alleviate locking and logging bottlenecks in the system, leaving page latching as the next potential problem. To tackle the page latching problem, we propose physiological partitioning (PLP). PLP applies logical-only partitioning, maintaining the desired properties of shared-everything designs, and introduces a multi-rooted B+Tree index structure (MRBTree) that enables the partitioning of the accesses at the physical page level. Logical partitioning and MRBTrees together ensure that all accesses to a given index page come from a single thread and, hence, can be entirely latch free; an extended design makes heap page accesses thread private as well. Moreover, MRBTrees offer an infrastructure for easy repartitioning and allow us to have a lightweight dynamic load balancing mechanism (DLB) on top of PLP. Profiling a PLP prototype running on different multicore machines shows that it acquires 85 and 68 % fewer contentious critical sections, respectively, than an optimized conventional design and one based on logical-only partitioning. PLP also improves performance up to almost 50 % over the existing systems, while DLB enhances the system with rapid and robust behavior in both detecting and handling load imbalances.

Keywords Physiological partitioning · PLP · Multi-rooted B+Trees · MRBtree · Dynamic load balancing · Re-partitioning

1 Introduction

Due to concerns over power draw and heat dissipation, processor vendors can no longer rely on rising clock frequencies or increasingly aggressive micro-architectural techniques to boost performance. Instead, they focus on parallelism by placing many independent processing cores in each chip. The resulting multicore designs require software to expose enough execution parallelism in order to exploit the abundant and rapidly growing hardware parallelism. However, this is not an easy task, especially given the high degree of hardware resource sharing common to multicore designs.

On-line transaction processing (OLTP) is a particularly complex data management application that needs to perform efficiently on modern hardware. It has been shown that conventional shared-everything OLTP systems may face significant scalability problems in highly parallel hardware [24]. There is increasing evidence that one source of scalability problems arises from the conventional transaction-oriented assignment of work policy that assigns each transaction to a single thread [40]. The transaction, along with the physical

Ippokratis Pandis and Ryan Johnson: work done while author affiliated with CMU and EPFL.

P. Tözün (✉) · A. Ailamaki
School of Computer and Communication Sciences,
École Polytechnique Fédérale de Lausanne,
Lausanne, VD, Switzerland
e-mail: pinar.tozun@epfl.ch

A. Ailamaki
e-mail: anastasia.ailamaki@epfl.ch

I. Pandis
IBM Almaden Research Center,
San Jose, CA, USA
e-mail: ipandis@us.ibm.com

R. Johnson
Department of Computer Science,
University of Toronto, Toronto, ON, Canada
e-mail: ryan.johnson@cs.utoronto.ca

Published online: 26 June 2012

arrangement of records within the data pages, determines what resources (e.g., records and pages) each thread will access. The random nature of transaction processing requests leads to unpredictable data accesses [40,48] that complicate resource sharing and concurrency control.

Such unpredictability favors pessimistic systems that clutter the transaction's execution path with many lock and latch acquisitions to protect the consistency of the data. These critical sections often lead to *contention* that limits scalability [24] and in the best case imposes a significant penalty to single-thread performance [19].

Following a different approach, shared-nothing systems deploy many independent database instances that collectively serve the workload [13,49]. In shared-nothing designs, the contention for shared data resources can be explicitly tuned (the database administrator (DBA) determines the number of processors assigned to each instance), potentially leading to superior performance as long as inter-instance communication can be minimized. The H-Store system takes this approach to the extreme, with single-threaded database instances that eliminate critical sections altogether [51]. However, shared-nothing systems physically partition the data and deliver poor performance when the workload triggers distributed transactions [11,20] or when skew causes load imbalance [11]. Repartitioning to rebalance load requires the system to physically move and reorganize all affected data. These weaknesses become especially problematic as partitions become smaller and more numerous in response to the multicore trend.

Recent work proposes logical-only partitioning [40] to address problems with conventional execution while avoiding the weaknesses of shared-nothing approaches. Logical-only partitioning assigns each partition to one thread; the latter manages the data locally without the overheads of centralized locking. However, purely logical partitioning does not prevent conflicts due to false sharing nor does it address the overhead and complexity of page latching protocols.

Ideally, we would like a system with the best properties of both shared-everything and shared-nothing designs: a centralized data store that sidesteps the challenges of moving data during (re)partitioning, and a partitioning scheme that eliminates contention and the need for page latches.

1.1 Dynamically balanced physiological partitioning

This paper presents *physiological partitioning (PLP)*, a transaction processing approach that partitions logically the physical data accesses. To alleviate the difficulties imposed by page latching and repartitioning, PLP uses a new physical access method, a type of multi-rooted B+Tree called *MRBTree*. Under PLP, a partition manager assigns threads to subtree roots of MRBTrees and ensures that requests distributed to each thread reference only the corresponding subtree. As a result, threads can bypass the partition mapping and their accesses to the subtree are entirely latch free. In addition, PLP can extend the partitioning down into the heap pages where non-clustered records are actually stored, eliminating another class of page latching (similar to shared-nothing systems). At the same time, the underlying MRBTree structure supports fast repartitioning and does not require distributed transactions when requests span partitions (like a shared-everything system).

To further exploit the MRBTrees, this paper also gives a lightweight yet effective dynamic load balancing and repartitioning mechanism, called *DLB*, on top of PLP. DLB uses the existing request queues of the partitions and employs a simple data structure, called *aging two-level histogram*, to collect information about the current access patterns and load in a workload to dynamically guide the decision on the partition maintenance.

1.2 Contributions and organization

This paper extends *physiological partitioning (PLP)* [42] with a dynamic load balancing mechanism, DLB, which is naturally integrated to PLP, and provides a more thorough evaluation of PLP and its limitations. The contributions and structure of the remaining of this paper is as follows:

We categorize the communication patterns, which clearly highlight the latent scalability bottlenecks. Using this categorization, we identify page latching as a lurking performance and scalability bottleneck in modern transaction processing systems, whose effect is proportional to the available hardware parallelism (Sect. 2).

We show that the need for page latching during accesses to both index and heap pages can be eliminated within a shared-everything OLTP system by deploying a design based on physiological partitioning (Sect. 3).

We demonstrate that even though partitioning-based OLTP systems have better throughput compared to non-partitioning based ones under uniform workloads, access skew can severely hurt performance in statically partitioned databases, rendering partitioning useless in many realistic workloads and underlining the need for dynamic repartitioning (Sect. 4.1).

We devise a cost model for repartitioning and exhibit that PLP provides a very good infrastructure for dynamic repartitioning, mainly due to its key component MRBTrees (Sect. 4.2), and we exploit this advantage by designing a lightweight yet effective dynamic load balancing and repartitioning mechanism, DLB, for PLP (Sect. 5).

We evaluate a prototype implementation of PLP integrated with DLB. PLP acquires 85 and 68 % fewer contentious critical sections per transaction than an optimized conventional design and a logically partitioned system, respectively, improving scalability and yielding up to almost 50 % higher

performance on multicore machines. In the meantime, the overhead of DLB is minimal in regular processing (in the worst case 8 %), and it achieves low response times in both detecting and balancing imbalances (Sect. 6).

While PLP advances the state-of-the-art design options for OLTP systems as discussed in Sect. 7, it has some limitations as well, which we detail in Sect. 8. Nevertheless, we conclude by promoting PLP as a very promising OLTP system design in the light of the upcoming hardware trends in Sect. 9.

2 Communication patterns

Traditional transaction processing systems excel at providing high concurrency or the ability to interleave multiple concurrent requests or transactions over limited hardware resources. However, as core counts increase exponentially, performance increasingly depends on execution parallelism or the ability for multiple requests to make forward progress simultaneously in different execution contexts. Even the smallest of serializations on the software side therefore impact scalability and performance [21]. Unfortunately, recent studies show that high concurrency in transaction processing systems does not necessarily translate to sufficient execution parallelism [24,25], due to the high degree of irregular and fine-grained communication they exhibit.

Proposals to tackle overhead and scalability bottlenecks fall into two general categories: (1) reducing the degree of communication and contention within shared-everything systems, relying on efficient communication via shared caches to keep synchronization overheads low; and (2) taking a shared-nothing approach [49], relying on the low-latency of multicore hardware to keep overheads manageable in spite of the challenges, which accompany distributed transactions and load balancing.

In this section, we first categorize the types of communication that can occur in an OLTP system, and from this point of view, we analyze the execution of a modern shared-everything system. Then, we revisit the debate between the shared-everything and shared-nothing approaches.

2.1 Types of communication

OLTP systems employ several different types of communication and synchronization. *Database locking* operates at the logical (application) level to enforce isolation and atomicity between transactions. *Page latching* operates at the physical (database page) level to enforce the consistency of the physical data stored on disk in the face of concurrent updates from multiple transactions. Finally, at the lowest levels, *critical sections* protect various code paths that must execute serially to protect the consistency of the system's internal state. Critical sections are traditionally protected by mutex locks, atomic instructions, etc. We note that locks and latches, which form a crucial part of the systems' internal state, are themselves protected by critical sections; analyzing the behavior of critical sections thus captures nearly all forms of communication in the DBMS.

Critical sections, in turn, fall into three categories depending on the nature of the contention they tend to trigger in the system. For example, pairs of threads that form producer–consumer pairs protect their communication with a critical section but cannot generate significant contention. We refer to these as *fixed* critical sections because contention is independent of the underlying hardware and depends only on the (fixed) number of threads that communicate. At the other extreme, *unscalable* critical sections have the highly undesirable tendency to affect most threads in the system. As hardware parallelism increases the degree of contention also increases and inevitably grows into a bottleneck. Making these critical sections shorter or less frequent provides a little slack but does not fundamentally improve scalability. Finally, Moir et al. [35] introduce the notion of *composable* critical sections; those having the property that multiple threads can aggregate their operations. Composable critical sections are highly resistant to contention because threads take advantage of queuing delays to combine their requests and drop out of the queue. The critical section is thus self-regulating: adding more threads to the system gives more opportunity for threads to combine rather than competing directly for the critical section.

2.2 Communication patterns in OLTP

As the previous section hints, the real key to scalability lies in converting all unscalable communication to either the fixed or composable type, thus removing the potential for bottlenecks to arise. The three left-most bars of Fig. 1 compare the number and types of critical sections executed by a conventional OLTP system and two others designed to reduce contention due to locking: Speculative Lock Inheritance [23] and data-oriented execution [40] (labeled as *SLI* and *Logi-*

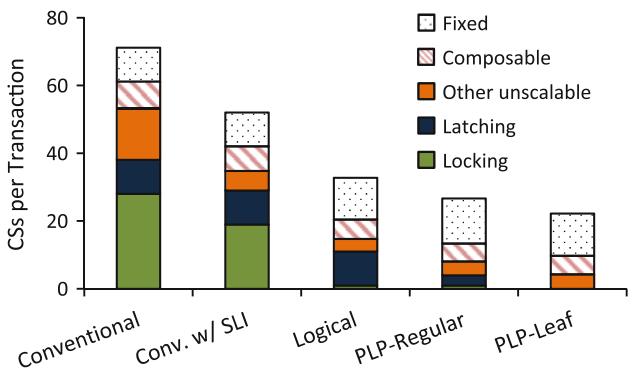

Fig. 1 Breakdown of the critical sections when running the TATP OLTP benchmark

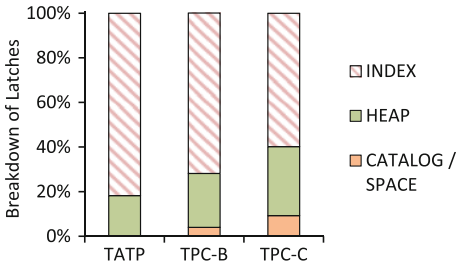

Fig. 2 Page latch breakdown for various OLTP benchmarks

cal-only, respectively). Each bar shows the number of critical sections entered during a mix of short transactions, categorized by the originating storage manager service (details in Sect. 6.1). Locking and latching form a significant fraction of the total communication for the baseline system. SLI achieves a performance boost by sidestepping the most problematic critical sections associated with the lock manager, but fails to address the remaining (still-unscalable) communication in that category. Logical partitioning, in contrast, eliminates nearly all types of locking, replacing both contention and overhead of centralized communication with efficient, fixed communication via message passing.

With locking removed, latching remains by far the largest source of critical sections. There is no predefined limit to the number of threads that might attempt to access a given page simultaneously, so page latching represents an unscalable form of communication, which should be either eliminated or converted to a scalable type. The remaining categories represent either fixed communication (e.g., transaction management), composable operations (e.g., logging [25]), or a minor fraction of the total unscalable component.

Examining page latching more closely, Fig. 2 decomposes the page latches acquired by three popular OLTP benchmarks into the different types of database pages: metadata, index pages, and heap pages. The majority of page latches (60–80 %) reside in index structures. Heap page latches are another non-negligible component, accounting for nearly all remaining page latches.

2.3 Physical versus logical partitioning

With the preceding characterization of communication patterns in mind, we now return to the question of logical partitioning (shared-everything) versus physical partitioning (shared-nothing). As its name suggests, logical partitioning eliminates unscalable communication at the logical level, namely database locking. However, it has little impact on the remaining communication, which arises in the physical layers and cannot be managed cleanly from the application level. Even when requests do not communicate at the application level, threads must acquire page latches and potentially perform other unscalable communication.

Shared-nothing systems [13,49] are an appealing design, giving the designer explicit control over the number of threads per instance. Thus, the contention on each component of the system can be controlled or even eliminated. However, such designs give up too much by eliminating all communication within the engine. Even the composable and fixed types of critical sections, which do not threaten scalability become problematic. For example, logging is not amenable to distribution [25], and physically partitioned systems either use a shared log [32] or eliminate it completely [51].

Perhaps the biggest challenge for shared-nothing systems arises with distributed transactions, due to requests accessing data from multiple physically distributed database instances. The scalable execution of distributed transactions has been an active field of research for the past three decades, with researchers from both academia and industry, persuasively arguing that they are fundamentally not scalable [8,20]. Furthermore, the performance of shared-nothing systems is very sensitive to imbalances in load arising from skew in either data or requests while non-partition aligned operations (such as non-clustered secondary indexes) may pose significant barriers to physical partitioning.

3 Physiological partitioning

We have seen how both logically and physically partitioned designs offer desirable properties, but also suffer from weaknesses that threaten their scalability. In this work, we therefore propose physiological partitioning (or PLP), a hybrid of the two approaches that combines the best properties of both. Like a physically partitioned system the majority of physical data accesses occur in a single-threaded environment, which obviate the need for page latching; like the logically partitioned system, locking is distributed without resorting to distributed transactions and load balancing requires almost no data movement.

3.1 Design overview

Each transaction in a typical OLTP workload accesses a very small subset of records via indexes (sequential scans are prohibitively expensive). PLP therefore centers around the indexing structures of the database. Figure 3 gives a high-level overview of a physiologically partitioned system. We adapt the traditional B+Tree [3] (top left of Fig. 3) for PLP by splitting it into multiple subtrees, each covering a contiguous subset of the key space (bottom of Fig. 3). A *partitioning table* becomes the new *root* and maintains the partitioning as well as pointers to the corresponding subtrees. We call the resulting structure a multi-rooted B+Tree (MRBTree). The MRBTree partitions the data but unlike a horizontally partitioned workload (top right of Fig. 3), all subtrees belong

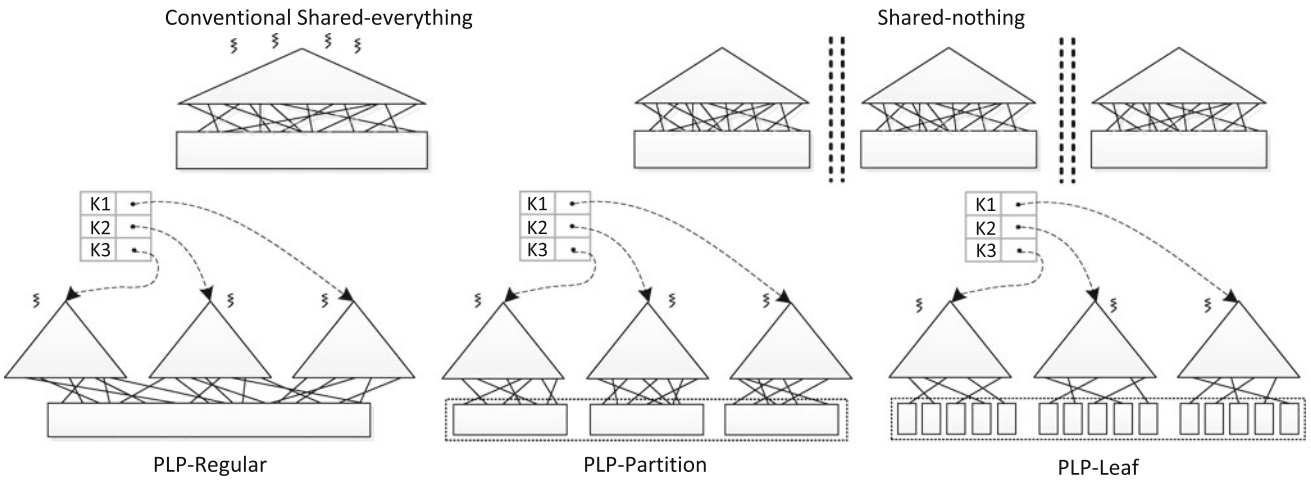

Fig. 3 The conventional shared-everything and shared-nothing designs and the PLP variations

to the same database file and can exchange pages easily; the partitioning, though durable, is dynamic and malleable rather than static.

With the MRBTree in place, the system assigns each subtree to a single thread, guaranteeing exclusive access for latch-free execution. A *partition manager* layer controls all partition tables and makes assignments to threads. The threads in PLP do not reference partition tables during normal processing, which might otherwise become a bottleneck. Instead, the partition manager ensures that all work given to a thread involves only data it owns.

The partition manager breaks transactions into directed graphs, passing each node to the appropriate thread and assembling the results into complete transactions. Since each table is assigned to have different set of worker threads, whenever a transaction touches more than one table it becomes a multisite transaction under PLP. However, multisite transactions are not expensive as in a shared-nothing system since PLP has a shared-everything setting.

All indexes in the system—primary, secondary, clustered, non-clustered—can be implemented as MRBTrees; data are stored directly in clustered indexes, or in tightly integrated heap file pages referenced by record ID (RID). When the system can infer partitions from secondary (non-clustered) index columns, the partition's thread manages them directly. The remaining (non-partition aligned) secondary indexes are accessed as in the conventional system, but each leaf entry records the associated fields used for the partitioning so that the result of each probe can be passed to its partition's owning thread for further processing.

3.2 Multi-rooted B+Tree

The "root" of an MRBTree is a partition table that identifies the disjoint subsets of the key range assigned to each subtree as well as a pointer to the root of each tree. Because the routing information is cached in memory as a ranges map by the partition manager, the on-disk layout favors simplicity rather than optimal access performance. We, therefore, employ a standard slotted page format to store key-root pairs. If the partitioning information cannot fit on a single page (for example, if the number of partitions is large or the keys are very long) the routing page is extended as a linked list of routing pages. In our experiments we have never encountered the need to extend the routing page, however, as several dozen mappings fit easily in 8 KB, even assuming rather large keys.

Record insertion (deletion) takes place as in a regular B+tree. When the key to insert (delete) is given, the ranges map routes it to the subtree that corresponds to the key range the key belongs to and the insert (delete) operation is performed as in a regular B+tree in that subtree. The other subtrees, ranges map, and the routing page do not get affected by the insert (delete) operation at all.

When deployed in a conventional shared-everything system, the MRBTree eliminates latch contention at the index root; fewer threads attempt to grab the latch for the same index root at a time. Partitioning also reduces the expected tree level by at least one, which reduces the index probe time. Further, the MRBTree can also potentially benefit systems that use shared-nothing parallelism in a shared-memory environment (e.g., possibly H-Store [51]).

3.3 Heap page accesses

In PLP a heap file scan is distributed to the partition-owning threads and performed in parallel. Large heap file scans reduce concurrency of OLTP applications and PLP has little to offer. Still, heap page management opens up an additional design option, since we can extend the partitioning of the

accesses at the heap pages. That is, when records reside in a heap file rather than in the MRBTree leaf pages, PLP can ensure that accesses to pages are partitioned in the same way as index pages. There are three options on how to place and access records in the heap pages, depicted in Fig. 3: (1) keep the existing heap page design (*PLP-Regular*); (2) each heap page keeps records of only one logical partition (*PLP-Partition*); and (3) each heap page is pointed by only one leaf page of the primary MRBTree (*PLP-Leaf*).

PLP-Regular simply keeps the existing heap page operations. Without any modification, the heap pages still need to be latched because they can be accessed by different threads in parallel. This may be acceptable because heap page accesses are not the biggest fraction of the total page accesses in OLTP (as low as 30 %, according to Fig. 2). Thus, there is room for significant improvement even if we ignore them. However, allowing heap pages to span partitions prevents the system from responding automatically to false sharing or other sources of heap page contention.

In PLP-Partition and PLP-Leaf the MRBTree and heap operations are modified so that heap page accesses are partitioned as well. The difference between the two is that in the former a heap page can be pointed by many leaf pages as long as they belong to the same partition, while in the latter a heap page is pointed by only one leaf page.

Both variations provide latch-free heap page accesses, but they suffer from some disadvantages. Forcing a heap page to contain records that belong to a specific partition causes fragmentation. In the worst case, each leaf has room for one more entry than fits in the heap page, resulting in nearly double the space requirement (Sect. 6.8 measures this cost). Further, in PLP-Leaf every leaf split must also move the records that are pointed by the new leaf page to a new heap page, increasing the overhead of record insertion (deletions are simple because a leaf may point to many heap pages). On the other hand, PLP-Partition by allowing multiple leaf pages from a partition to share a heap page, forces the system to reorganize potentially significant numbers of heap pages with every repartitioning. Significant reorganization costs go against the philosophy of physiological partitioning, so we favor PLP-Leaf.

The two extensions impose one additional piece of complexity: During record insertion, the system must identify the correct MRBTree entry before selecting a heap page for the record. Because the storage management layer is completely unaware of the partitioning strategy (by design), it must make callbacks into the upper layers of the system to identify an appropriate heap page for each insertion.

Similarly, a partition split may split heap pages as well, invalidating the record IDs of migrated records. The storage manager, therefore, exposes another callback so the metadata management layer can update indexes and other structures that reference the stale RIDs. We note that when PLP-Leaf splits leaf pages during record insertion, the same kinds of record relocations arise and use the same callbacks.

3.4 Page cleaning

Page cleaning cannot be performed naively in PLP. Conventionally there is a set of page cleaning threads in the system that are triggered when the system needs to clean dirty pages (for example, when it needs to truncate log entries). Those threads may access arbitrary pages in the buffer pool, which breaks the invariant of PLP where a single thread can access a page at each point of time.

To handle the problem of page cleaning in PLP each thread does the page cleaning for its logical partition. Each logical partition has an additional input queue for system requests and the page cleaning requests go to that queue. The system queue has higher priority than the queue of completed actions. Their execution won't be delayed by more than the execution time of one action (typically very short). In addition, because page cleaning is a read-only operation, the thread can continue to work (and even re-dirty pages) during the write-back I/O.

3.5 Benefits of physiological partitioning

Under physiological partitioning, each partition is permanently locked for exclusive physical access by a single thread, which then handles all the requests for that partition. This allows the system to avoid several sources of overhead, as described in the following paragraphs.

Latching contention and overhead. Though page latching is inexpensive compared to acquiring a database lock, the sheer number of page latches acquired imposes some overhead and can serialize B+Tree operations as transactions crab down the tree during a probe. The problem becomes more acute when the lower levels of the tree do not fit in memory, because a thread that fetches a tree node from disk holds a latch on the node's parent until the I/O completes might be preventing access to 80–100 other siblings, which may well be memory-resident. Section 6.3 evaluates a case where latching becomes expensive for B+Tree operations and how PLP can eliminate this problem by allowing latch-free accesses on index pages.

False sharing of heap pages. One significant source of latch contention arises when multiple threads access unrelated records that reside on the same physical database page. In a conventional system false sharing requires padding to force problematic database records to different pages. PLP variations that allow latch-free heap page accesses achieve the same effect automatically (without the need of expensive tuning) as it splits hot pages across multiple partitions. Section 6.3 evaluates this case as well.

Serialization of structural modification operations. The traditional ARIES/KVL indexes [33] allow only one structural modification operation (SMO), such as a leaf split, to occur at a time, serializing all other accesses until the SMO completes. Partitioning the tree physically with MRB-Trees eases the problem by distributing SMOs across sub-trees (whose roots are fixed) without having to apply more complicated protocols, as such those described in [22,34]. The benefits of parallel SMOs are apparent in the case of insert-heavy workloads, which we evaluate in Sect. 6.5.

Repartitioning. In PLP, repartitioning can occur at a higher level in the partition manager and therefore can be latch free as well; the partition manager can simply quiesces affected threads until the process completes. Moreover, it requires very few pointer updates and data movement as we discuss in Sect. 4.2. Therefore, it can be performed very efficient as shown in Sect. 6.9.

Code complexity. Finally, with all latching eliminated, the code paths that handle contention and failure cases can be eliminated as well, simplifying the code significantly. To the extend that the index can be substituted with a much simpler implementation. For example, a huge source of complexity in traditional B+Trees arises due to the sophisticated protocols that maintain consistency during an SMO in spite of concurrent probes from other threads. The simpler code not only is more efficient but also easier to maintain. In this paper, we did not attempt the code refactoring required to exploit these opportunities, and the performance results we report are therefore conservative. But, we note that B+Tree probes are the most expensive remaining component of PLP. Thus, we expect significant performance improvements if, for example, we substitute the B+tree implementation of our prototype with a cache-conscious [43,44] or prefetching-based B+-tree [10].

4 Need and cost of dynamic repartitioning

Although partitioning is an increasingly popular solution for scaling up the performance of database management systems, it is not the panacea since there are many challenges associated with it. One of these challenges is the system behavior under skewed and dynamically changing workloads, which is the rule rather than an exception in real settings (e.g., the *slashdot effect*).

This section shows how even mild access skew can severely hurt performance in a statically partitioned database, rendering partitioning useless in many realistic workloads (Sect. 4.1). Then it exhibits that PLP provides an adequate infrastructure for dynamic repartitioning, mainly because it is based on physiological partitioning and because of its usage of MRBTrees (Sect. 4.2). The low repartitioning cost facilitates the implementation of a robust yet lightweight dynamic load balancing mechanism for PLP, which is presented in the following section (Sect. 5).

4.1 Static partitioning and skew

In general, one of the disadvantages of partitioning-based transaction processing designs is that they are vulnerable to skewed and dynamically changing workloads, in contrast to shared-everything systems that do not employ any form of partitioning and tend to suffer less. Unfortunately, skewed and dynamically changing workloads are the rule rather than an exception in transaction processing. Thus, it is imperative for partitioning-based designs to alleviate the problem of skewed and dynamically changing accesses.

To exhibit how vulnerable partitioning-based systems are to skew, Fig. 4 plots the throughput of a non-partitioned (shared-everything) system and a statically partitioned system when all the clients in a `TATP` database submit the `GetSubscriberData` read-only transaction [38]. Initially the distribution of requests is uniform to the entire database. But at time point 10 (s) the distribution of the load changes with 50 % of the requests being sent to 30 % of the database (see Sect. 6.1 for experimental setup details). As we can see from the graph, initially and as long as the distribution of requests is uniform the performance of the non-partitioned system is around 15 % lower than the partitioned one. After the load change the performance of the non-partitioned system remains pretty much the same (at around 325 Ktps), while the performance of the partitioned system drops sharply by around 35 % (from 375 Ktps to around 360 Ktps). The drop in the performance is severe even though the skew is not that extreme; easily a higher fraction of the requests could go to a smaller portion of the database, for example following the 80–20 rule of thumb where the 80 % of the accesses go to only 20 % of the database.

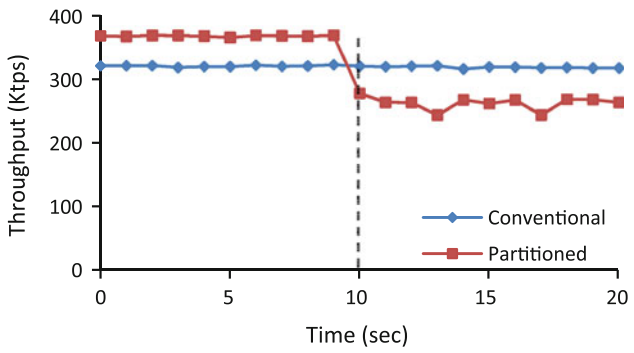

Fig. 4 Throughput of a statically partitioned system when load changes at runtime; at time $t = 10$, 50 % of the requests are sent to 30 % of the database

There are two ways to attack the problem of skewed access in partitioning-based transaction processing systems: *proactively* by configuring the system with an appropriate initial partitioning scheme; and *reactively* by using a dynamic balancing mechanism. Starting with the appropriate partitioning configuration is key. If the workload characteristics are known a priori, previously proposed techniques [11,45] can be used to create effective initial configurations. If the workload characteristics are not known, then simpler approaches like round-robin, hash-based, and range-based partitioning can be used [13]. As time progresses, however, skewed access patterns gradually lead to load imbalance and lower performance, as the initial partitioning configuration eventually becomes useless no matter how carefully it was chosen. Thus, it is far more important and challenging to *dynamically balance the load* through *repartitioning* based on the observed, and ever changing, access patterns. A robust dynamic load balancing mechanism should eliminate any bad choices made during initial assignment.

4.2 Repartitioning cost

A dynamic load balancing mechanism would be useless if the cost of repartitioning in a partitioning-based transaction processing system is high. The lower the cost of repartitioning, the more frequently the system can trigger load balancing procedures and the faster it can react to load changes. This subsection models the cost of repartitioning for a shared-nothing (physically partitioned) system and the three PLP variations to highlight the clear advantage of PLP-Regular and PLP-Leaf. It also describes the way to perform repartitioning for the three PLP designs.

The basic case of repartitioning is when a partition needs to split into two. Thus, for all the PLP variations and the shared-nothing design our repartitioning cost model calculates the number of records and index entries that have to be moved, the number of update/insert/delete operations on the indexes, the number of pointer updates on the index pages and the routing page, and the remaining number of read operations that have to be performed when a partition is split into two. We also discuss merging two partitions but do not give as detailed cost model.

Let's assume that there is a heap file (table) with an index on it, which in the case of PLP it is an MRBTree. When a partition needs to split into two, that means that a subtree in the index needs to split into two as well. In that case we define as: h the height of the tree; n the number of entries in an internal B+Tree node; m_i the number of entries to be moved from the B+Tree at level i; and M the number of records in the heap file that have to be moved.

The number of read operations during a key value search in the B+Tree is omitted since it is the same for all the systems (a binary search at each level from root to leaf).

4.2.1 Splitting non-clustered indexes

The first case we consider, is when the heap file that needs to be re-partitioned has a unique non-clustered primary and a secondary index and the data are partitioned based on the primary index key values.

PLP-Regular. The cost of repartitioning in PLP-Regular is very low. Only a few index entries need to move from one subtree of the MRBTree index(es) to another newly created subtree. Algorithm 1 shows the procedure that needs to be executed to split an MRBTree subtree. First, we need to find the leaf page that the starting key of the new partition should reside (Lines 4–8 in Algorithm 1). Let's assume that there are m_1 entries that are greater than or equal to the starting key on the leaf page where the slot for this key is found. All needs to be done is to move these m_1 entries on that leaf page to a newly created (MRBTree) index node page and this procedure has to repeat as the tree is traversed from this leaf page to the root (Lines 9–13 in Algorithm 1). It is not necessary to move any entry from the pages that keep the key values greater than the ones in the leaf page containing the starting key. Setting the previous/next pointers of the pages at the boundaries of the old and new partitions is sufficient. Finally, a new entry to the routing page should be added for the new partition.

The overall cost is given in the first row of Table 1. The cost model in Table 1 describes the worst case scenario for PLP-Regular. If the starting key of the new partition is in one of the internal index node pages, there is no need to move any entries from the pages that are below this page because the moved entries from the internal node page already have pointers to their corresponding child pages; resulting in fewer reads, updates, and moved entries.

PLP-Leaf. The partition splitting cost related with the MRBTree index structure is the same as in PLP-Regular. But, as mentioned in Sect. 3.3, in addition to modifying the index

Algorithm 1 Splitting an MRBTree subtree.

1: {binary-search routine used below performs binary search to find the key on the $page$. If an exact match for the key is found, $found$ is returned as true and the function returns the slot for the key on the $page$. Otherwise, $found$ is false and the function returns which slot on the $page$ the key should reside.}
2: $page = root$
3: $found = false$
4: **while** $page != NULL$ & $!found$ **do**
5: $slot =$ binary-search$(page, key, found)$
6: $slots.push(slot)$
7: $pages.push(page)$
8: $page = page[slot].child$
9: **while** $nodes.size > 0$ **do**
10: $slot = slots.pop()$
11: $page = pages.pop()$
12: Create $page_{new}$
13: Move starting from $slot$ at $page$ to $page_{new}$

Table 1 Repartitioning costs for splitting a partition into two

System	#Records moved (M)	Primary index					Secondary index
		#Entries moved	#Reads	#Pages read	#Pointer updates	Changes	Changes
PLP-Regular	–	$\sum_{k=1}^{h} m_k$	–	–	$2 \times h + 1$	–	–
PLP-Leaf	m_1	$\sum_{k=1}^{h} m_k$	M	1	$2 \times h + 1$	M updates	M updates
PLP-Partition	$m_1 + \sum_{l=0}^{h-2}(n^{h-l-1} \times (m_{h-l} - 1))$	$\sum_{k=1}^{h} m_k$	M	$1 + \frac{M-m_1}{n}$	$2 \times h + 1$	M updates	M updates
Shared-Nothing	$m_1 + \sum_{l=0}^{h-2}(n^{h-l-1} \times (m_{h-l} - 1))$	–	M	$1 + \frac{M-m_1}{n}$	–	M inserts M deletes	M inserts M deletes
PLP (clustered)	m_1	$\sum_{k=2}^{h} m_k$	–	–	$2 \times h + 1$	–	M updates
Shared-Nothing (clustered)	$m_1 + \sum_{l=0}^{h-2}(n^{h-l-1} \times (m_{h-l} - 1))$	–	–	–	–	M inserts M deletes	M inserts M deletes

structure, when repartitioning in PLP-Leaf, we also have to move records from the heap file to new heap pages. Figure 5 shows the three-step process for splitting a partition to two in PLP-Leaf.

The height of the subtree is two and the dark slot in Fig. 5a indicates the slot that contains the leaf entry with the starting key of the new partition. Figure 5b shows that a new subtree is created as a result of the split. Those two steps are the same with the repartitioning process in PLP-Regular.

In PLP-Leaf, however, we also have to move the records at the heap file that belong to the new partition to a new set of heap data pages. Algorithm 2 shows the pseudo code for updating the heap pages upon a partition split in PLP-Leaf (and PLP-Partition). The dark records on the heap pages in Fig. 5b indicate those records that belong to the new partition (subtree) and need to move. Those records are pointed by the m_1 leaf page entries that moved to the newly created subtree. Thus, in the worst case m_1 records have to move (Lines 4–7 in Algorithm 2). Since the index is non-clustered, we have to scan these m_1 entries in order to get the RIDs of the records to be moved and spot their heap pages. The result of the split after the records are moved is shown in Fig. 5c. Whenever a record moves its RID changes. Thus, once all the records are moved, all the indexes (primary and secondary) need to update their entries (Line 8 in Algorithm 2).

Algorithm 2 Splitting heap pages in PLP-Leaf and PLP-Partition.

```
1: leaf = leftmost leaf node
2: Create page_new
3: while leaf != NULL {Omit for PLP-Leaf} do
4:   for all t pointed by leaf_current do
5:     if page_new does not have space then
6:       Create page_new
7:     Move t to page_new
8:     Update pointers at all the secondary indexes
9:   leaf = leaf.next {Omit for PLP-Leaf}
```

The cost for repartitioning in PLP-Leaf is given in the second row of Table 1. This cost, again, illustrates the worst case scenario. If the starting key of the new partition is found in one of the internal nodes, then no record movement has to be done since there will be no leaf page splits and the constraint of having all heap pages pointed by only one leaf page is already preserved. Moreover, even if the key is found on the leaf page, we might not have to move all the records that are specified by the model above. If all the records on a heap page are pointed only by leaf entries of the new partition, then these records can stay on that heap page.

PLP-Partition. In PLP-Partition, the process for splitting the index structure is the same as in PLP-Regular and PLP-Leaf. Therefore, it is omitted from Fig. 6, which shows the

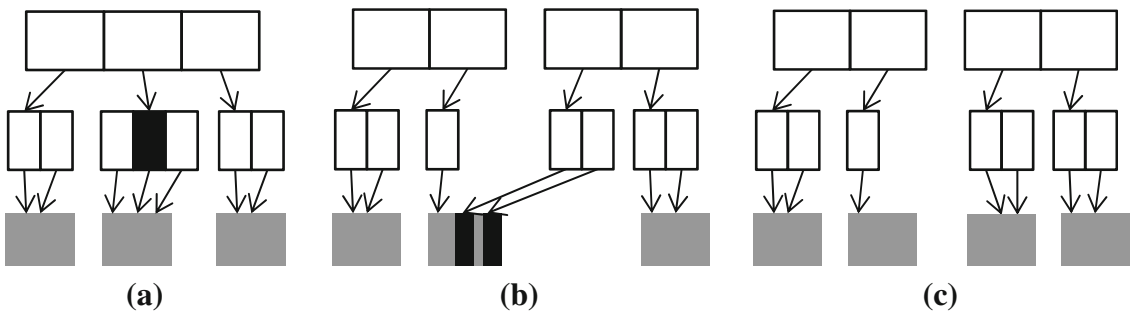

Fig. 5 Example of splitting a partition in PLP-Leaf, which is a three-step process

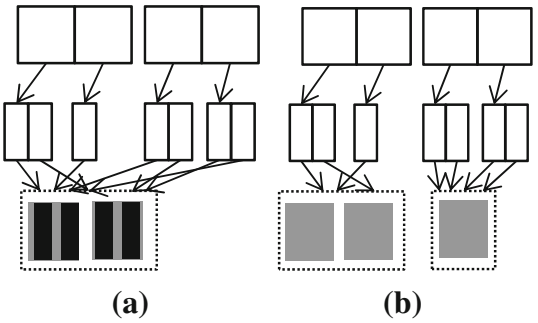

Fig. 6 Splitting a partition when PLP-Partition is used

rest of the process for splitting a partition into two in PLP-Partition.

In the worst case, in PLP-Partition we may have to move records from all the heap pages that belong to the old partition. Those records are indicated with the dark rectangles in the heap pages of Fig. 6a. The number of records to be moved is equal to the number of entries that are on the leaf pages of the new subtree. As in PLP-Leaf, the RIDs of the records are retrieved with an index scan of the newly created subtree, the records are moved to new heap pages and they get new RIDs, and all the indexes are updated with the new RIDs after the record movement is completed (shown in Lines 3–9 in Algorithm 2). The result of the partitioning is shown in Fig. 6b; while the cost model for PLP-Partition is given in the third row of Table 1.

Shared-Nothing. In a shared-nothing system, the cost for the record movement is equal to the worst case of PLP-Partition since the entire old partition needs to be scanned for records that belong to the new partition. In addition, the cost of index maintenance may be prohibitively expensive.

That is, in a shared-nothing system each record move across partitions results to a deletion of an index entry (or entries if there are multiple indexes) from the old partition and an insertion of an index entry to the new partition, in contrast to the PLP variant where every record move is a result of a few MRBTree updates. The cost of index maintenance when repartitioning shared-nothing systems sometimes can be prohibitive. In order to avoid the index maintenance, a common technique is to drop and bulk-load the index from scratch upon every repartition. For shared-nothing systems that employ replication, this procedure has to be repeated for all the partition replicas. The repartitioning cost for one replica in a shared-nothing system is given as in the fourth row of Table 1. Given how expensive repartitioning can be, shared-nothing systems are reluctant to frequently triggering repartitioning.

4.2.2 Splitting clustered indexes

Let's consider the case where we have a unique clustered primary index and a secondary index, and the data partitioning is done using the primary index key columns. In this setup, no heap file exists, since the primary index contains the actual data records rather than RIDs, and the three PLP variations are equivalent, because their differences lie on how they treat the records in the heap pages.

When the actual records are part of the clustered primary index, the cost of record movement for PLP is equal to the number of leaf page entries that need to move while the cost of primary index maintenance is equal to the entry movements in the internal nodes of the MRBTree index. The cost model is given in the fifth row of Table 1.

On the other hand, the repartitioning cost for the shared-nothing system is similar to the non-clustered case. Because there is not a common index structure and data need to move from the index of the one partition to the other. The only difference is there is no need to scan the leaf pages to get the RIDs of the records to be moved since the leaf pages have the actual records. Therefore, the repartitioning cost model for a replica is given as it is in the last row of Table 1.

4.2.3 Moving fewer records

With some additional information we can actually move fewer data during repartitioning with the cost of increased number of reads. For example, in PLP-Partition instead of directly moving all the records that belong to a new partition, we can scan all the index leaf pages to be split and collect information for all the records. With this information, we can determine whether a heap page has more records that belong to the old partition or the new partition and act accordingly. That is, if a heap page has more records that belong to the new partition, we can move out of the page the records that belong to the old partition. The number of reads while scanning the leaf pages can easily become a bottleneck in disk-resident databases, due to the number of I/O operations that have to be performed. On the other hand, in in-memory databases or systems that use flash storage devices, the I/O bottleneck can be prevented [9] and the above mentioned technique can reduce the amount of data movement during repartitioning. This technique, unfortunately, cannot be used in a shared-nothing system because the pages of the two partitions do not share the same storage space.

4.2.4 Example of repartitioning cost

Table 2 gives an example of the repartitioning cost for the different systems under consideration based on the cost model given in Table 1. In this example, a partition, which contains 433 MB of 100-byte data records in a heap file is split in half. We assume that there is a primary index of height 3 with 170 32-byte entries on each page. The first four rows of the table assume there is a unique non-clustered primary index and a secondary index in the system, whereas for the

Table 2 Repartitioning costs when splitting a partition with 466 MB data in half

System	Records moved	Primary index				Secondary index
		Entries moved	#Pages read	#Pointer updates	Changes	Changes
PLP-Regular	–	8 KB	–	7	–	–
PLP-Leaf	8.3 KB	8 KB	1	7	85 U	85 U
PLP-Partition	233 MB	8 KB	14,365	7	2.44M U	2.44M U
Shared-Nothing	233 MB	–	14,365	–	2.44M I + 2.44M D	2.44M I + 2.44M D
PLP (clustered)	8.3 KB	5.3 KB	–	7	–	85 U
Shared-Nothing (clustered)	233 MB	–	–	–	2.44M I + 2.44M D	2.44M I + 2.44M D

(*U* updates, *D* deletes, *I* inserts)

last two rows there is a unique clustered primary index and a secondary index. The cost for the shared-nothing system is just for one replica (if we assume that it uses replication for durability). For the PLP variations the number of moved records represents the worst case scenario.

As Table 2 shows, the PLP variations, except for PLP-Partition, move very few records compared to the shared-nothing one. In the worst case, PLP-Partition moves the same number of records as the shared-nothing system. For the clustered index case, PLP is cheaper to repartition than the shared-nothing system, both in terms of record movement and index maintenance. When we calculate the corresponding costs for a larger heap file with an index of height 4, the repartitioning cost for the shared-nothing system (and PLP-Partition) becomes prohibitive.

4.2.5 Cost of merging two partitions

For any PLP variation, merge operation only requires index reorganization and no data movement. During the index reorganization, there are three cases to consider; (1) when two subtrees have the same level, (2) when the subtree with lower key values (T_l) has a higher level than the other subtree, and (3) when the subtree with higher key values (T_h) has a higher level than the other subtree.

When the two subtrees to be merged have the same level, the entries of T_h's root are appended at the end of the entries of T_l's root. Since the entries of the root page have information about the pointers to the interior nodes, copying the entries of the root page is sufficient for this merge operation. In this case the cost of the merge operation only depends on the number of entries in the root page of T_h. If the number of entries destined to the new root exceeds the page capacity, an SMO happens and a new root page is created, the same way a page split happens after a record insert.

When T_l is taller than T_h, T_l is traversed down to one level higher than the level of T_h. Then an entry is inserted at the right-most node of this level that points to T_h and has the key value equal to the starting key of the key range of T_h. Therefore, the cost of the merge operation is only a tree traversal, which depends on the level difference between the two trees and an insert operation in this case.

When T_h is taller, the merge operation is very similar to the second case and the cost is the same. T_h is traversed down to one level higher than the level of T_l and instead of the right-most node, the left-most node gets the entry that points to T_l and has the key value equal to the starting key of the key range of T_l.

After the delete operation, the partition table is updated according to the new key range and its corresponding subtree root page id.

In a Shared-Nothing system, however, we have to move all the records from one partition to the other and insert the corresponding index entries at the resulting partition. Therefore, the cost of the merge operation is proportional to the number of records in a partition and its way higher than the merge cost for any PLP variation.

We conclude that, in contrast to shared-nothing systems, the PLP-Regular and PLP-Leaf designs provide low repartitioning costs that allow frequent repartitioning attempts and facilitate the implementation of responsive and lightweight dynamic load balancing mechanisms. We present one such mechanism in the next section.

5 A dynamic load balancing mechanism for PLP

At the high level, any dynamic load balancing mechanism performs the same functionality. During normal execution it has to observe the access patterns and detect any skew that causes load imbalance among the partitions. Once the mechanism detects the troublesome imbalance, it triggers a repartition procedure. It is very important for the detection mechanism to incur minimal overhead during normal

operation and to not trigger repartitioning when it is not really needed. After the mechanism decides to proceed to a repartition, it needs to determine a new partitioning configuration, so that the load is again uniformly distributed. This decision depends on various parameters, such as the recent load of each partition and the available hardware parallelism. Finally, after the new configuration has been determined, the system has to perform the actual repartitioning. The repartitioning should be done in a way that minimizes the drop in performance and the duration of this process.

Thus, any dynamic load balancing mechanism that we build on top of PLP (or any partitioning-based system in general) should; (a) perform lightweight monitoring, (b) make robust decisions on the new partition configuration, and (c) repartition efficiently when such decision is made. We have already shown that PLP provides the infrastructure for efficient repartitioning, in Sect. 4. In this section, we present techniques for lightweight monitoring and decision making. The overall mechanism is called *DLB*.

5.1 Monitoring

DLB needs to monitor some indicators of the system behavior and based on the collected information to decide: (a) when to trigger a repartition operation and (b) what the new partitioning configuration should be. Candidate indicators can be the overall throughput of the system, the frequency of accesses in each partition, and the amount of work each partition should do.

There is need for DLB to continuously collect information of multiple indicators. For example, let's consider that DLB monitors only the overall throughput of the system and raises flags when changes in throughput are larger than a threshold value. If the initial partitioning configuration of the system was not optimal (e.g., with load imbalance among partitions) then its throughput would be low without fluctuations—the effect caught when monitoring only the throughput, and the monitoring would fail. Or there could be uniform drops or increases in the incoming request traffic, which would trigger unnecessary repartitioning. Thus, DLB needs to maintain additional information about the load of each partition. In addition, the information about the throughput is not useful for the component that decides on the new configuration (presented in Sect. 5.2). Thus, DLB needs to collect and maintain information about the load not only across partitions, but also within each partition.

To that end, DLB uses the length of the request queue of each partition and a two-level histogram structure that employs *aging*. The histogram structure is depicted on Fig. 7. To monitor the differences in the load across partitions, DLB monitors the number of requests waiting at each partition's request queue. To have accurate information about the load distribution within each partition, in addition to the one

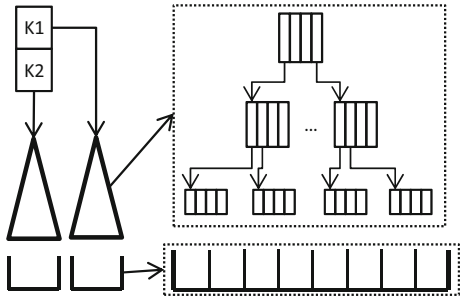

Fig. 7 Two-level histogram on MRBtrees

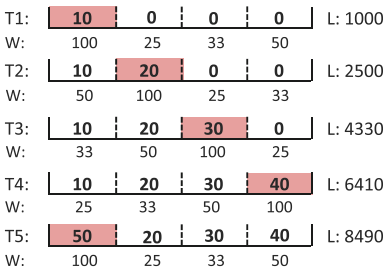

Fig. 8 The aging algorithm example

bucket it maintains for each partition (left side of the figure), the histogram has sub-buckets on ranges within each partition's key range (shown on the right side of the figure). The number of sub-buckets within each partition is tunable and determines the monitoring granularity.

DLB frequently checks whether the partition loads are balanced or not. The load of each partition is calculated based on an aging algorithm. Each bucket in the histogram is implemented as an array of *age-buckets*, shown on Fig. 8. At each point of time there is one active age-bucket. When a record is accessed, the active age-bucket of the sub-bucket of the range where the record belongs to increments by one. At regular time intervals the age of the histogram increases. Whenever the age of the histogram increases, the next age-bucket is reset and starts to count the accesses.

When calculating the load of a sub-bucket in the histogram, the recent age-buckets are given more weight than the older ones. More specifically, if a sub-bucket consists of A age-buckets, the load for the ith age-bucket is l_i, and the current age-bucket is the cth bucket, then we calculate the total load L for the sub-bucket as follows:

$$L = \sum_{i=c}^{A+c-1} \frac{100 \times l_{imod(A)}}{(i-c+1)}.$$

Figure 8 shows an example of the aging algorithm, when the load to a particular sub-bucket increases by 10 for five consecutive time intervals ($T1$ to $T5$). W is the weight of each age-bucket and L is the load value of this sub-bucket at each interval, calculated by the formula above.

Because they are both lightweight, DBL very frequently monitors the throughput and the length of the request queues. On the other hand, the histograms are analyzed only whenever an imbalance is observed. The overall monitoring mechanism does not incur much overhead and it also provides adequate information for DLB to decide on the new partitions.

5.2 Deciding new partitioning

The algorithm DLB employs for reconfiguring the partition key-ranges is highly dependent on the request queues and two-level aging-histogram structure discussed previously. First we describe the algorithm that determines the partitioning configuration within a single table and then we consider the case when we decide the partitioning across all tables.

Deciding the partitioning within a single table. To describe the algorithm, let N be the total number of partitions, and Q_i be the number of requests at the request queue of the ith partition. Then, the ideal number of requests for "each partition's queue" is:

$$Q_{\text{ideal}} = \sum_{i=1}^{N} \frac{Q_i}{N}.$$

Knowing Q_{ideal}, we have to decide on the ideal data access load for each partition. Let L_i be the aging load of the ith partition, which can be calculated as the sum of the aging loads of its sub-buckets. We have to calculate the ideal data access load for partition i, LI_i, based on the ideal request load and how much request load, Q_i, each L_i creates. Therefore, LI_i is:

$$LI_i = \frac{Q_{\text{ideal}} \times L_i}{Q_i}.$$

Because the granularity on the load information is determined by the number of sub-buckets in the histogram, it is difficult for DLB to achieve precise ideal loads. Therefore, DLB only tries to approximate the precise ideal value. Algorithm 3 sketches how the new key-ranges are assigned. DLB iterates over all partitions except for the last one. While the estimated load L_i at a partition is less than $LI_i - t$ for some t value, it moves the range of the leftmost sub-bucket from the $(i+1)$th partition to ith. Similarly, while the load at a partition is larger than $LI_i + t$, it moves the range of right-most sub-bucket from the ith partition to $(i+1)$th. If the moved sub-bucket causes a significant change in the calculated load (more than $2 \times t$), then this sub-bucket is substituted by a larger number of sub-buckets to observe that range in finer-granularity.

Figure 9 has an example of how Algorithm 3 is applied. In the example, there are three partitions on a table and Fig. 9 shows the two-level histogram for each partition. The first-level of the histogram tracks down the number of accesses to a partition's range, which is 40 units in this example. The second-level of the histogram, the 4 sub-buckets, keeps the number of accesses to sub-ranges in a partition, which is 10 units in this example. The higher bar in a sub-bucket indicates that the sub-range that corresponds to that sub-bucket has more load. Initially, each partition has equal key-ranges, shown in the left-hand side of Fig. 9. If we assume that each partition has to perform equal amount of work per request, the loads in this configuration are not balanced among the partitions. Therefore, the repartition manager triggers repartitioning. Based on Algorithm 3 the new partitions are decided by moving around the sub-buckets to create almost-equal loads among the partitions. The result is shown on the right-hand side of Fig. 9; the most loaded regions end up in partitions with smaller ranges, like the second partition in Fig. 9, and the lightly loaded regions are merged together.

Algorithm 3 Calculating ideal loads.

```
1: for i = 1 → N − 1 do
2:   while L_i < LI_i − t do
3:     Move leftmost sub-bucket range from i + 1 to i
4:     L_i ⇐ L_i + L_subbucket
5:     if L_i > LI_i + t then
6:       Distribute sub-bucket range into μ sub-buckets
7:   while L_i > LI_i + t do
8:     Move rightmost sub-bucket range from i to i + 1
9:     L_i ⇐ L_i − L_subbucket
10:    if L_i < LI_i − t then
11:      Distribute sub-bucket range into μ sub-buckets
```

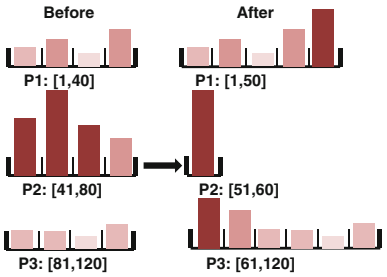

Fig. 9 Example of how to decide on the new partition ranges

Deciding the number of partitions of each table. The algorithm presented above is just for one table and assumes that the number of partitions before and after the repartitioning operation does not change. To determine how many partitions a table should have is another issue and requires knowledge on all of the tables in the database.

In our setting, initially, the number of partitions for a table is determined automatically to be equal to the number of hardware contexts supported by the underlying machine. To find what the number of partitions for a table should be dynamically, based on the workload trends; let T be the number of tables, N_{total} be the upper limit on the total number of partitions for the whole database, Q_i be the total number of

requests for table i, N_i be the number of partitions for table i, QT_{avg} be the average number of requests for all the tables, N_{avg} be the average number of partitions for a table, and $\#CTX$ be the total number of available hardware contexts supported by the machine that executes the transactions run on this database.

Based on the initial total number of partitions, we define N_{total} as: $N_{\text{total}} = T \times \#CTX$. As a result, N_{avg} will be: $N_{\text{avg}} = \frac{N_{\text{total}}}{T} = \#CTX$. The QT_i values are known from the request queues, and therefore, QT_{avg} can be calculated as: $QT_{\text{avg}} = \frac{\sum_{i=1}^{T} QT_i}{T}$. The goal is to find the N_i values, which can be derived from the following formula: $\frac{QT_{\text{avg}}}{N_{\text{avg}}} = \frac{QT_i}{N_i}$. Using the formulas and algorithm presented above, DLB efficiently decides on the new partitioning configuration.

5.3 Using control theory for load balancing

In our prototype, the system immediately tries to adjust to a new configuration, once a target load value is determined for each partition. Thus, there is always the danger of over-fitting, especially for the workloads that observe access skew with frequently changing hot-spots. Since repartitioning is not expensive for PLP (except for PLP-Partition), it can repartition again very quickly to alter the bad effects of a previous bad partitioning choice. Rather than directly aiming to reach the target load, a more robust technique would be to employ control theory while converging to the target load [31]. Control theory can increase the robustness of our algorithm, prevent the system from repartitioning unnecessarily and/or resulting with wrong partitions, and reduce the downtime faced by PLP-Partition during repartitioning. Nevertheless, it is orthogonal with the remaining infrastructure, and it could be easily integrated in the current design. The prototype implementation does not employ control theory techniques. But the evaluation, presented next, shows that DLB allows PLP to balance the load effectively.

6 Evaluation

The evaluation consists of four parts. In the first part we measure how useful PLP can be. In particular; Sect. 6.2 quantifies how different designs impact page latching and critical section frequency, Sect. 6.3 examines how effectively PLP reduces latch contention on index and heap page latches, and Sect. 6.4 shows the performance impact of those changes. In the second part we try to quantify any overheads related to PLP. To do that we measure PLP's behavior in challenging workloads that seem to not fit well with physiological partitioning, such as transactions with joins (Sect. 6.6) and secondary index accesses that can be aligned with the partitioning or not (Sect. 6.7). In addition, Sect. 6.8 inspects the fragmentation overhead of the three PLP variations. In the third part (Sect. 6.5) we quantify how useful MRBTrees can be also for conventional and logically partitioned systems. In the last part of the evaluation, we measure the overhead and effectiveness of the dynamic load balancing mechanism of PLP (Sects. 6.9, 6.10). Finally, In Sect. 6.11, we highlight the key conclusions of the whole evaluation.

6.1 Experimental setup

To ensure reasonable comparisons, all the prototypes are built on top of the same version of the Shore-MT storage manager [24], incorporate the logging optimizations of [25], and share the same driver code.

We consider five different designs: (a) An optimized version of a conventional, non-partitioned system, labeled as *Conventional*. This system employs speculative lock inheritance [23] to reduce the contention in the lock manager; (b) *Logical-only* is a data-oriented transaction processing prototype [40] that applies logical-only partitioning; (c) *PLP or PLP-Regular* prototypes the basic PLP variation. This variation accesses the MRBTree index pages without latching; (d) *PLP-Partition* extends PLP-Regular, so that one logical partition "owns" each heap page, allowing latch-free both index and heap page accesses; (e) *PLP-Leaf* assigns heap pages to leaves of the primary MRBTree index, also allowing latch-free index and heap page accesses. In addition, we experiment with the PLP variations with the dynamic load balancing mechanism integrated. We label those systems with a "-DLB" suffix (PLP-Reg-DLB, PLP-Part-DLB, and PLP-Leaf-DLB).

All experiments are performed on two machines: an x64 box, with four sockets of quad-core AMD Opteron 8356 processors, clocked at 2.4 GHz and running Red Hat Linux 5; and a Sun UltraSPARC T5220 server with a 64-core Sun Niagara II chip clocked at 1.4 GHz and running Solaris 10. Due to unavailability of a suitably fast I/O sub-system, all experiments are with memory-resident databases. But the relative behavior of the systems will be similar with larger databases.

6.2 Page latches and critical sections

First we measure how PLP reduces the number of page latch acquisitions in the system. Figure 10 shows the number and type of page latches acquired by the conventional, the logically partitioned, and two variations of the PLP design; PLP-Regular and PLP-Leaf. Each system executes the same number of transactions from the `TATP` benchmark. PLP-Regular reduces the amount of page latching per transaction by more than 80 %; while PLP-Leaf reduces the total further to roughly 1 % of the initial page latching. The remaining latches are associated with metadata and free space management.

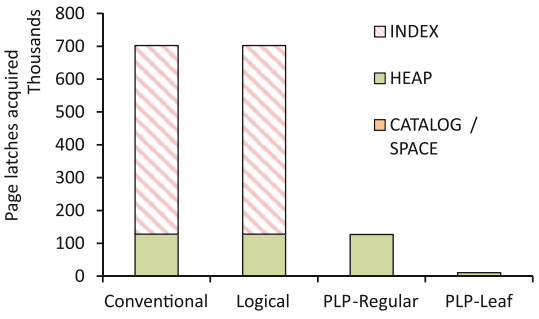

Fig. 10 Page latches acquired by different designs in TATP

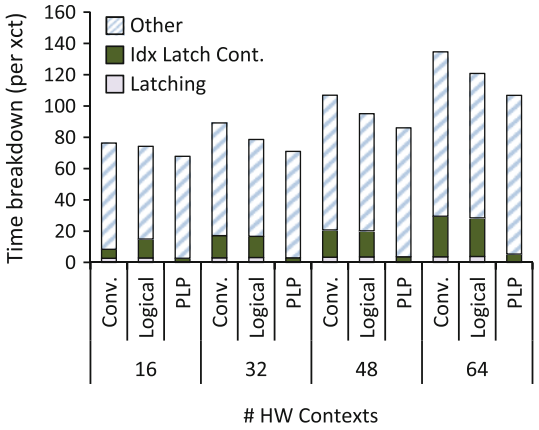

Fig. 11 Time breakdown per transaction in an insert/delete-heavy benchmark

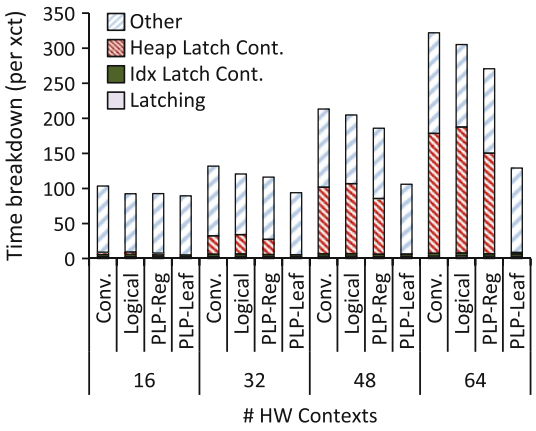

Fig. 12 Time breakdown per transaction in TPC-B with false sharing on heap pages

The two right bars of Fig. 1 compare total critical section entries of PLP versus the conventional and logically partitioned systems. The two PLP variants eliminate the vast majority of lock- and latch-related critical sections, leaving only metadata and space management latching as a small fraction of the critical sections. Transaction management, which is the largest remaining component, mostly employs fixed-contention communication to serialize threads that attempt to modify the transaction object's state. Similarly, the buffer pool-related critical sections are mostly due to the communication between cleaner threads, which again do not impact scalability. Overall, PLP-Leaf acquires 85 and 65 % fewer contentious critical sections than the conventional and logically partitioned systems, respectively.

6.3 Reducing index and heap page latch contention

Having established that PLP effectively reduces the number of page latch acquisitions and critical sections, we measure what is the impact of that change in the time breakdown.

Figure 11 shows the impact in the transaction execution time as PLP eliminates the contention on index page latches. The graph gives the time breakdown per transaction for the different designs as increasing number of threads run an insert/delete-heavy workload on the TATP database. In this benchmark, each transaction makes an insertion or a deletion to the CallFwd table, causing page splits and contention for the index pages that lead to the records being inserted/deleted. As Fig. 11 shows, the conventional and the logically partitioned systems experience contention on the index page latches. They both spend 15–20 % of their time waiting, while PLP eliminates the contention achieving proportional performance improvements.

Figure 12 gives the time breakdown per transaction when we run TPC-B benchmark [53]. In this experiment we do not pad records to force them onto different pages. Transactions often wait for others because the record(s) they update happen to reside on latched heap pages. The conventional, logically partitioned, and PLP-Regular all suffer from false sharing of heap pages. At high utilization this contention wastes more than 50 % of execution time. On the other hand, PLP-Leaf is immune, reducing response time by 13–60 % and achieving proportional performance improvement. In a way, PLP-Leaf provides automatic and more robust padding for the workloads that require manual padding in the conventional system to reduce contention on the heap pages.

Figure 13 has the time breakdown per transaction when 16 and 40 hardware contexts are utilized by the conventional, logically partitioned, and PLP-Partition systems when they run a slightly modified version of StockLevel transaction of TPC-C benchmark. StockLevel contains a join, and in this version, 2,000 tuples are joined. We see that the conventional system wastes 20–25 % of its time in contention in the lock manager and for page latching. Interestingly, the logically partitioned system eliminates the contention in the lock manager, but this elimination is not translated to performance improvements. Instead the contention is shifted and aggravated to the page latches. On the other hand, PLP eliminates the contention both in the lock manager and for page latches and achieves higher performance.

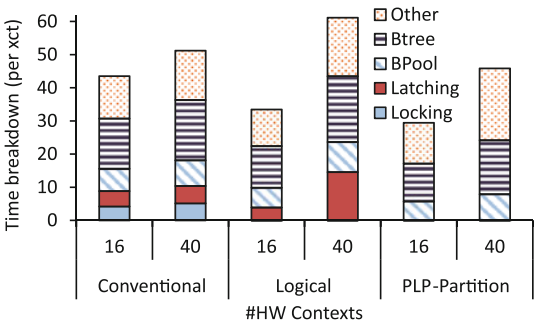

Fig. 13 Time breakdown for the `StockLevel` transaction in `TPC-C` when 2,000 tuples joined

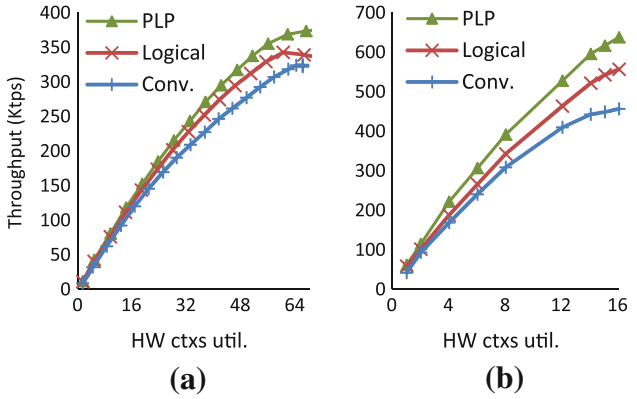

Fig. 14 Throughput of the `GetSubscriberData` transaction in two multicore machines. **a** Sun Niagara II **b** 4-socket Quad x86_64

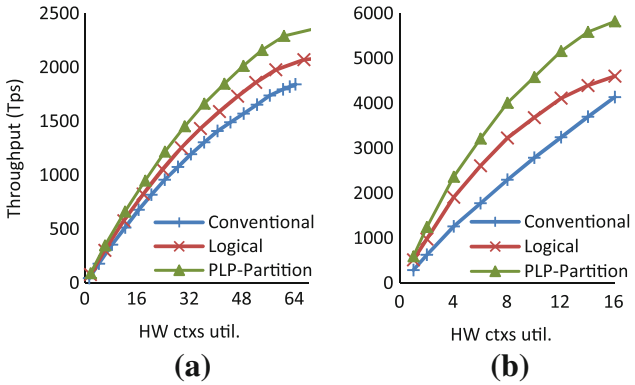

Fig. 15 Throughput of the `StockLevel` transaction in two multicore machines. **a** Sun Niagara II **b** 4-socket Quad x86_64

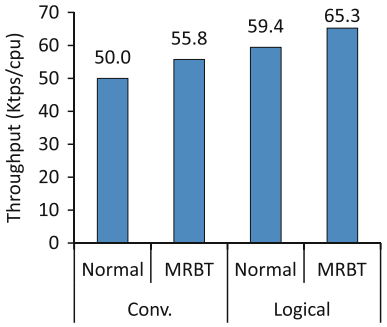

Fig. 16 Performance of a conventional and a logically partitioned system in `TATP`

6.4 Impact on scalability and performance

Since PLP effectively reduces the contention (and the time wasted) to acquire and release index and heap page latches, we next measure its impact on performance and overall system scalability. Figures 14 and 15 show the throughput of the three main designs under comparison as we increase hardware utilization of the two multicore machines. The workloads consists of clients that repeatedly submit the `TATP-GetSubscriberData` and `TPCC-StockLevel` transactions, respectively, which are read-only and ideally should impose no contention whatsoever.

As expected, PLP shows superior scalability, evidenced by the widening performance gap with the other two systems as utilization increases. For example, from Fig. 15b, we see that for `StockLevel`, logically partitioned system delivers an 11 % speedup over the baseline case in the 4-socket Quad x64 system. PLP delivers an additional 26 % over logical partitioning or nearly 50 % over the conventional. The corresponding improvements in the Sun machine's slower but more numerous cores are 13 and 34 %. Note that eight cores of the x64 machine match the fully loaded Sun machine, so the latter does not expose bottlenecks as strongly in spite of its higher parallelism. A significant fraction of the speedup actually comes from the MRBTree probes, which are effectively one level shallower, since threads bypass the "root" partition table node during normal operation.

6.5 MRBTrees in non-PLP systems

The MRBTree can improve performance even in the case of conventional systems in three ways. First, since it effectively reduces the height of the index by one level, each index probe traverses one fewer node and hence it is faster. Second, any possible delay due to contention on the root index page is also reduced roughly proportionally with the number of subtrees. We see the effect of those two in Fig. 16, which highlights the difference in the peak performance of the conventional and the logically partitioned system when they run with and without MRBTrees. The workload is the `TATP` benchmark. In both cases the improvement in performance is in the order of 10 %.

Third, MRBTrees allow each subtree to have a structure modification operation (SMO) in flight at any time, in

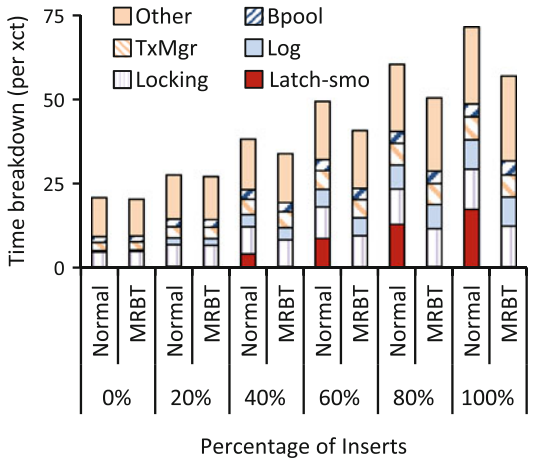

Fig. 17 Time breakdown of conventional transactions when parallel SMOs are allowed with MRBTrees

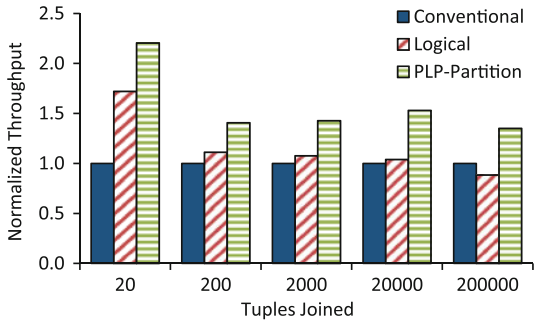

Fig. 18 Maximum throughput normalized to *Conventional* for `StockLevel` when the 4-socket Quad x86_64 is 100 % utilized

contrast to traditional B+Trees that can have only one SMO in flight. Consequently, in workloads with high entry insertion (deletion) rates, the MRBTree improves performance by parallelizing the SMOs. Figure 17 shows the time breakdown of the conventional system with and without MRBTrees as we run a microbenchmark that consists of either a record probe or insert, and we increase the percentage of inserts. Without MRBTrees, the system spends an increasing amount of time blocked waiting for SMOs to complete as the insertion rate increases. When MRBTrees are used, there is no time wasted waiting for SMOs and performance improves by up to 25 %. Overall, there are compelling reasons for systems other than PLP to adopt MRBTrees.

6.6 Transactions with joins in PLP

Next we turn our attention to workloads that seem to not fit well with physiological partitioning. First, we inspect how PLP behaves with transactions that has join operations.

To evaluate the performance of PLP on transactions with joins, we slightly modified the `StockLevel` transaction from the `TPC-C` benchmark [54] to determine the number of tuples joined. In its un-modified version, `StockLevel` joins 200 tuples between two tables. We created different versions of the transaction where 20, 200, 2,000, 20,000, and 200,000 tuples are joined. For each different number of tuples joined, Fig. 18 plots the maximum throughput the conventional, the logically partitioned, and the PLP-Partition systems achieve, normalized to the maximum throughput of the conventional. The three systems achieve their maximum throughput when the 4-socket Quad x64 machine is 100 % utilized, which means that there is no significant scalability bottlenecks. Figure 18 shows that the PLP variation achieves higher performance than the conventional system regardless of the number of tuples joined. When only 20 tuples are joined PLP achieves 2.1x higher performance than conventional, while when 200 K tuples are joined PLP achieves 33 % higher performance. PLP achieves higher performance because it eliminates the contention for page latches, as Fig. 13 illustrates. That is in contrast to the logically partitioned system, which for large number of tuples joined performs lower than conventional.

6.7 Secondary index accesses

Non-clustered secondary indexes are pervasive in transaction processing, since they are the only means to speed up transactions that access records using non-primary key columns. Nevertheless, secondary index accesses pose several challenges to PLP, which we explore in Fig. 19. We break this analysis into two cases: when the secondary index is aligned with the partitioning scheme and when it is not.

We conduct an experiment where we modify `TATP`'s `GetSubscriberData` transaction to perform a range scan on the secondary index that is built on the names of the `Subscribers` and we control the number of matched records. In the original transaction only one `Subscriber` is found. In the modified version, we probe for 10, 100, 1,000, and 10,000 `Subscribers`, even though index scans for thousands of records are not typical in high-throughput transactional workloads.

If the secondary index columns are a subset of the routing columns, then the secondary index is aligned with the partitioning scheme. In that case, a secondary index scan may return a large number of matched RIDs (record ids of entries that match the selection criteria) from several partitions. All the executors need to send the probed data to a coordination point where an aggregation of the partial results takes place. As the range of the index scans become larger (or the selectivity drops), this causes a bottleneck due to excessive data transfers. When the secondary index is not aligned with the partitioning scheme, then on top of the above mentioned bottleneck there is also an important overhead. This overhead is because each record probe becomes a two step process,

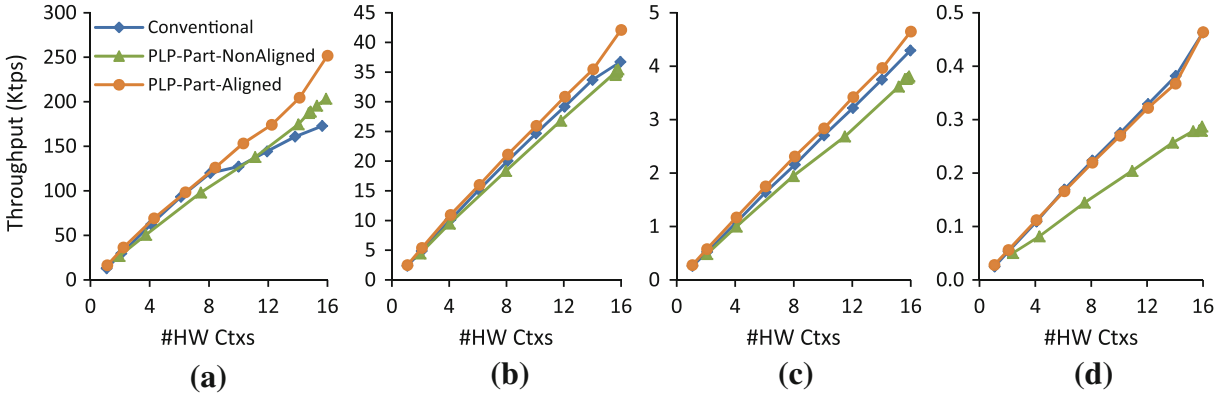

Fig. 19 Performance on aligned and non-aligned secondary index scans. **a** Range = 10, **b** range = 100, **c** range = 1,000, **d** range = 10,000

where the secondary index probe is done by one thread conventionally and then requests from the appropriate executor threads to retrieve the selected records.

Figure 19 compares the performance of *Conventional* system with *PLP-Part-Aligned*, which performs partitioning aligned secondary index accesses, and *PLP-Part-NonAligned*, which performs non-partitioning aligned secondary index accesses, as more hardware contexts are utilized in the system. *PLP-Part-Aligned* improves performance over *Conventional* by 46, 14, 8, and 1 %, respectively, for ranges 10, 100, 1,000, 10,000. On the other hand, even though *PLP-Part-NonAligned* improves performance by 11 % when 10 records are scanned, for larger ranges it hinders performance. *PLP-Part-NonAligned* is 3, 11, and 38 % slower than *Conventional* for ranges 100, 1,000, and 10,000, respectively.

As expected, the performance improvement for *PLP-Part-Aligned* gets smaller as the range of the index scan increases. However, as long as the index scans of partitioning-aligned secondary indexes are selective and touch a relatively small number of records, PLP provides decent performance improvement. For *PLP-Part-NonAligned*, however, such workloads are very unfriendly, though unless the scan range is over 1,000 records it is not disastrous.

6.8 Fragmentation overhead

PLP-Partition and PLP-Leaf, create some fragmentation on the heap file since they change the regular heap file structure (see Sect. 3.3). Given the increased number of data pages due to fragmentation, we expect the heap file scan times to increase proportionally.

Figure 20 shows the ratio between the number of pages used in the three PLP variations and the conventional system as we increase the database size. The x-axis shows the total size of the database when each record is 100B (left-hand side of the graph) and 1000B (right-hand side of the graph).

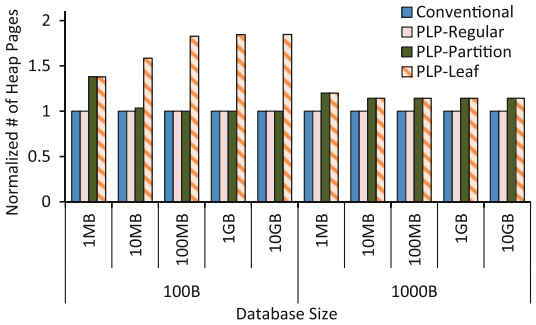

Fig. 20 Space overhead of the three PLP variations

The y-axis is the ratio between the number of pages used in each design and the conventional system. The conventional system has one partition, where the PLP variations have 100 and 10 partitions for the cases where record size is 100B and 1000B, respectively. The heap page size is 8 KB. As expected, PLP-Regular does not create any fragmentation since it maintains the regular heap file format. For PLP-Partition, the amount of fragmentation becomes negligible as the database size increases for small records. However, PLP-Leaf uses up to 80 % more heap pages than a conventional system for the same case creating a visible fragmentation on the heap file. On the other hand, as we increase the record size, the fragmentation decreases because each heap page is able to keep fewer records, and thus, the amount of empty space on each heap page is reduced.

Figure 21 shows the time to scan the heap file for each PLP variation compared to the conventional system as we increase the size of the database. The setup is same as in Fig. 20 when the record size is 100B. The size of the buffer pool is 4 GB for each measurement. From Fig. 21, the fragmentation cost of PLP-Leaf does not significantly increase the file scan time when there are no I/O operations performed (from 1 MB to 1 GB) because the total number of records that are scanned is the same. However, for the larger database

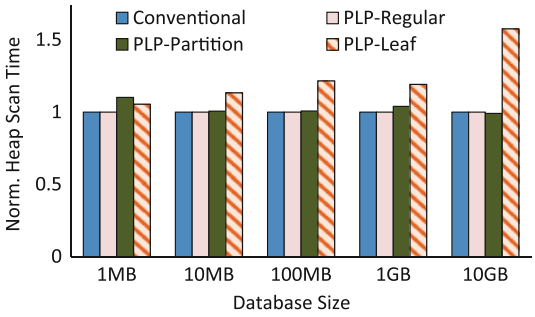

Fig. 21 Overhead of PLP variations during file scan

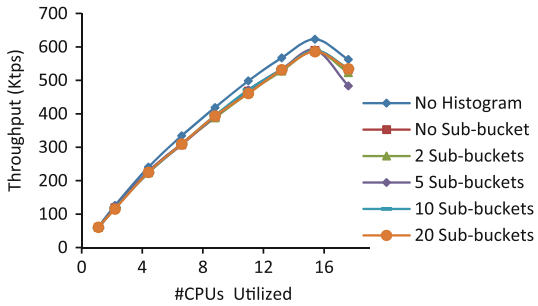

Fig. 22 Overhead of updating histogram for DLB under normal operation

(10 GB), PLP-Leaf increases the heap file scan time by 60 % since there are more I/O requests.

Overall, among the PLP variations, only PLP-Leaf may introduce some significant fragmentation when a heap page can keep many database records. As the number of records a heap page can keep decreases, this cost becomes less significant. We also note that PLP is a design optimized for high-performing transactional applications, where entire heap file scans are rare.

6.9 Overhead and effectiveness of DLB

In this section we first quantify the overhead of the dynamic load balancing mechanism (DLB) under normal operation. Then we measure how quickly and effectively DBL re-acts against skew and load imbalances. All the experiments use the `GetSubscriberData` transaction from `TATP` benchmark. We picked this transaction since it probes a record from the `Subscribers` table, which provides 10,000 tuples per scaling factor (and per partition in our experiments). Therefore, the records that have to change partitions after repartitioning will be sufficient enough to understand the record movement cost among the different PLP variations.

6.9.1 Overhead in normal operation

Under normal operation, DLB should impose minimal overhead. DLB's monitoring component performs three operations: it maintains the histograms with access information, it continuously monitors the throughput, and it periodically analyzes the request queues of the worker threads for load imbalances. Since this monitoring is performed by a separate thread, it should not affect the throughput of the system at all unless all the CPUs in the system are utilized by the threads executing transactions. Therefore, the main source of overhead for DLB is updating the histogram.

Figure 22 shows the overhead caused by updating the aging histogram for each data access. Since the number of threads that try to update the histogram increases, as we utilize more CPUs, the overhead of updating the histogram increases as well. On the other hand, increasing the number of sub-buckets does not have much effect. We note that histogram is not a source of contention since each partition has their own sub-buckets that they update. Therefore, the overhead in updating the histogram purely comes from the extra work that a partition's worker thread has to perform while updating the histogram.

Overall, the monitoring component of DLB is fairly lightweight. On average histogram updates cause 6 % drop in throughput compared to the system running without a histogram and maximum drop is 7–8 %. Considering that the transaction we execute in our system is a read-only transaction, we actually evaluate the worst case behavior here. For a transaction with updates, the number of transactions executed per second and hence the number of data accesses would be lower. Fewer data accesses would cause fewer updates in histogram and therefore less overhead.

6.9.2 Reacting to load imbalances

In order to evaluate how effectively DLB handles load imbalances, we execute the same experiment as the one in Fig. 4. The PLP variations (PLP-Regular, PLP-Reg-DLB, PLP-Part-DLB, and PLP-Leaf-DLB) use 64 partitions, apply aging in every 1 s, and the load difference threshold value t is 10 %. Initially the requests are distributed uniformly and at time point 10 (s), 30 % of the database starts to receive 50 % of the requests.

As Fig. 23 shows, the change in the access pattern causes a 30 % drop in the throughput of *PLP-Regular*, making its performance worse than the performance of the non-partitioned *Conventional* system. On the other hand, the DLB-integrated PLP variations quickly detect the skew and bring the performance back to the pre-skew levels in less than 10 s. In particular, 2 s after the change in the access pattern, DLB has already decided on the new partitioning configuration, and around 8 s later it has performed 126 repartition operations (63 splits and 63 merges). The throughput has some spikes for a short time after repartitioning, but in the end settles down.

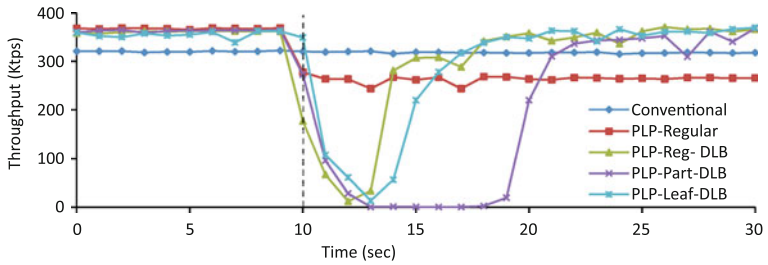

Fig. 23 Dynamic load balancing when at time $t = 10$, 50 % of the requests are sent to 30 % of the database

In *PLP-Reg-DLB*, very few index entries are updated, leading to a shorter dip in throughput during repartitioning. *PLP-Leaf-DLB* experiences an almost equally short dip. *PLP-Part-DLB* suffers a much longer dip. For the statically partitioned PLP, Fig. 23 has only the results for the statically partitioned PLP-Regular since the drop in throughput is almost the same for the other two statically partitioned PLP variations (PLP-Partition and PLP-Leaf).

DLB triggers a global repartitioning process that affects all the partitions in the system. PLP-Regular and PLP-Leaf can handle this process very well. However, such global repartitioning is not suitable for PLP-Partition. PLP-Partition is the closest to a shared-nothing system in terms of repartitioning cost since it reorganizes a large number of heap pages (see Sect. 4.2). Therefore, its non-optimal behavior with DLB is as expected.

6.9.3 Speeding up accesses to hot spots

When DLB is effective, the "hot" regions end up to narrow partitions. The indexes for these partitions are shallower and provide shorter access times for the "hot" records. In addition, "hot" records that could previously belong to the same partition, due to their key proximity, end up to different partitions. Figure 24 illustrates graphically the impact of DLB on the ranges of 10 partitions before and after a repartitioning. The area within the rectangular region highlights the "hot" range; it is 10 % of the total area that receives the 50 % of the total load. Initially, labeled *Before*, the system has equal-length range partitions. After DLB kicks in and repartitioning completes, labeled *After*, the "hot" region has shorter-length range partitions while the not-so-loaded regions have larger-length partitions.

Table 3 shows the average index probe time (in microseconds) for a hot record as we increase the skew. For this experiment we use a single table with 640,000 records for a total size of around 1 GB. There is an index of this table, with 8 KB pages and the primary key is an integer (4B). When there are 10 equal-range partitions, the height of each partition's subtree is 3. Each row in the table shows the average access time of a randomly picked record from a "hot" region that gets 50 % of all the requests, as the range of the

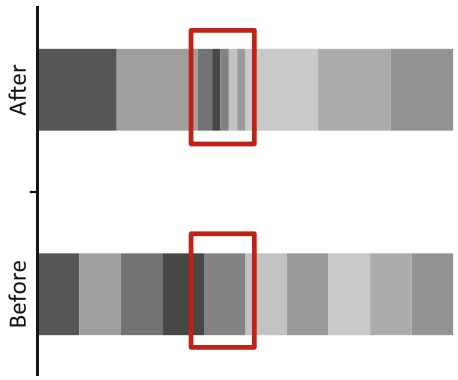

Fig. 24 Partitions before and after the repartitioning

Table 3 Average index probe times (in microseconds) for a hot record, as skew increases

Skewed region (%)	Before skew	After skew	After repartitioning
50	69	67	65
20	67	66	63
10	69	66	62
5	68	64	61
2	68	64	60

"hot" region decreases—and the skew increases. The first column ("Before skew") shows the average access time when the requests are uniformly distributed. The second column ("After skew") shows the average access time when DLB is disabled and the request distribution is skewed. The third column has the average access time after DLB kicked in and completed a repartitioning.

As Table 3 shows, the access times for the randomly picked record is lower after we set the skew. This is probably due to some caching effect since the record is accessed more frequently when there is skew in data accesses. However, the access time after repartitioning is the shortest since the height of the subtree in the new "hot" partition is 2 whereas in the old partition it was 3 (the height of the subtrees for the other partitions remains as 3).

Table 4 shows the number of finished requests for the "hot" record after the skew and after DLB's repartitioning. Before repartitioning fewer requests are satisfied for the picked record because its partition is highly loaded with requests

Table 4 Average record probes per second for a hot record, as skew increases

Skewed region (%)	After skew	After repartitioning
50	13	13
20	7	29
10	7	73
5	32	108
2	63	155

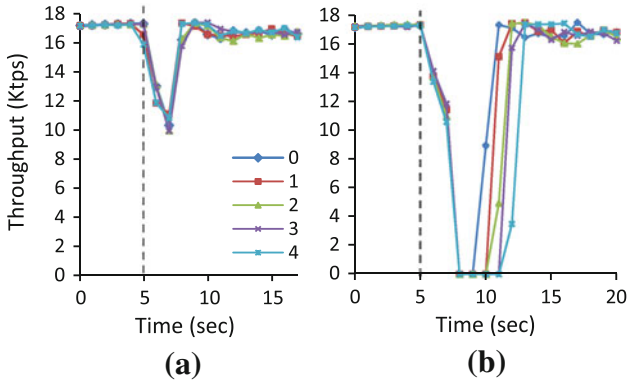

Fig. 25 Overhead of updating secondary indexes during repartitioning. At time $t = 5$ 50 % of the requests are sent to only 10 % of the database, which triggers repartitioning. **a** PLP-Leaf, **b** PLP-Partition

for other records in the same "hot" partition range. DLB distributes the "hot" range between multiple shorter-range partitions. Therefore, a single partition can serve more requests for the "hot" record. This results in small throughput increase after repartitioning in Fig. 23.

6.10 Overhead of updating secondary indexes for DLB

In PLP-Leaf and PLP-Partition, whenever a record moves every non-clustered index of the table needs to be updated with the record's new RID (see Sect. 3.3). In this section, we measure the overhead of updating the secondary indexes during repartitioning.

Figure 25 shows the effect of repartitioning on throughput as we increase the number of secondary indexes for a table for PLP-Leaf and PLP-Partition. For this experiment we use the `Subscribers` table of the `TATP` database. Initially, there are 2 partitions of 320,000 records each that receive uniform requests. After 5 s, 50 % of the requests are sent to only 10 % of the table and DLB triggers a repartitioning. We measure the throughput of the system as we increase the number of secondary indexes on the table, from none up to 4 secondary indexes.

Figure 25a shows that the overhead for PLP-Leaf while updating the secondary is relatively low, because very few or no records need to be moved. On the other hand, the overhead for PLP-Partition is much higher. PLP-Partition has to move more records and update more entries in the secondary indexes. Therefore, repartitioning in PLP-Partition takes longer time as we increase the number of secondary indexes for a table.

6.11 Summary

As the experimental results show, PLP, successfully, manages to eliminate two major sources of unscalable critical sections in conventional shared-everything systems; locking and latching. In addition, it provides a good infrastructure for easy repartitioning and dynamic load balancing. It is important to note that each PLP variation has its drawbacks. For example, PLP-Leaf comes with some fragmentation (Sect. 6.8) and PLP-Partition cannot repartition efficiently (Sects. 6.9.2, 6.10). Considering the long lasting throughput drops during repartitioning for the PLP-Partition, we favor PLP-Leaf for workloads that need dynamic load balancing. If the workload does not heavily suffer from heap page latching, but only index page latching, then PLP-Regular is definitely a great design choice because it neither has fragmentation nor faces long and sharp drops in throughput during repartitioning.

7 Related work

The related work can be categorized in three; analyzing and reducing the critical sections in DBMSs, partitioned B+trees and concurrency control mechanisms, and dynamic load balancing and repartitioning.

7.1 Critical sections

The complexity and overheads of database management systems are well-known. For example, [19] shows that, even in a single-threaded OLTP system, logging, locking, latching, and buffer pool accesses contribute roughly equal overheads and together account for the majority of machine instructions executed during a transaction. Our previous work shows that these overheads become scalability burdens in multicore hardware [24]. PLP eliminates entire categories of serializations, along with the corresponding bottlenecks.

In the shared-everything arena, recent proposals for speculative lock inheritance [23] and data-oriented transaction execution [40] minimize the need for interaction with a centralized lock manager. Where speculative lock inheritance allows the system to spread lock operations across multiple transactions to reduce contention, data-oriented systems replace the central lock manager with thread-local lock management. Reducing lock contention with data-oriented execution is also studied for data-streams' operators [12] by making

threads delegate the work on some data to the thread that already holds the lock for that data and move to the next operation in their queues.

Other proposals tackle the weakness posed by the centralized log manager; [25] presenting a scalable log buffer and [9] exploiting flash technology to reduce logging latencies. These proposals show even seemingly pervasive forms of communication can be reduced or sidestepped to great effect. However, none of them addresses physical data accesses involving page latching and buffer pool, the other two major overheads in the system, which PLP eliminates.

Oracle RAC [39], with Cache-Fusion [28], allows database instances in the shared-disk cluster to share their buffer pools and avoid accesses to the shared-disk. It can also partition the data to reduce both logical and physical contention on a particular portion of the data. However, it does not enforce each partition to be accessed only by a single thread. Therefore, it does not eliminate physical latch contention while accessing pages from the shared-cache as much as PLP does.

As discussed previously, shared-nothing [13,49,51] systems have an appealing design that eliminates critical sections altogether. However, they struggle both pro-actively to reduce the need to execute distributed transactions through efficient partitioning [11] as well as re-actively to reduce overheads when distributed transactions cannot be avoided [26]. On the other hand, PLP, in addition to eliminating a big portion of the unscalable critical sections, offers a less costly way of load balancing and communication for distributed transactions since partitions share the same memory space.

7.2 B+trees and alternative concurrency control

Alternatives to traditional B+tree concurrency control are studied to allow multiple SMOs at the same time [22,34]. The MRBTree index structure provides an alternative to such techniques, allowing concurrent SMOs with less code complexity. However, these techniques can be implemented alongside with MRBTrees to achieve concurrency within a partition, should that be desirable for a conventional system. As an addition to these techniques MRBTrees also allow multiple root split operations in parallel. Several earlier works propose B+Trees having multiple roots to reduce contention due to locking [16,37]. However, again none of these proposals targets physical latch contention in the system.

In addition, there are latch-free B+tree implementations that use alternative synchronization methods. CO B-Tree [5] uses load-linked/store-conditional (LL/SC) instead of latching to synchronize operations on a B+tree. However, it does not eliminate contention on the B+tree. PALM [46] eliminates both page latching and contention on the B+trees by using Bulk Synchronous Parallel model. However, it has to perform B+tree operations in batches in order to exploit this technique, which might not be desirable all the time and harder to integrate within a database management system.

Finally, optimistic and multiversioning concurrency control schemes [6,27,29] may improve concurrency by resolving conflicts lazily at commit time instead of eagerly blocking them at the moment of a potential conflict. When conflicts are rare this allows the system to avoid the overhead of enforcing database locks. On the other hand, if the conflicts occur frequently the performance of the system drops rapidly, since the transaction abort rate is high. Moreover, there is work that compares the concurrency control schemes in database systems. Notable is the work by Agrawal et al. [2], while the book of Bernstein et al. [7] and Thomasian's survey [52] are good starting points for the interested reader. On the other hand, the focus of PLP is on the contention for latches rather than the concurrency scheme used.

We also note that there is a large body of work on cache-conscious index implementations (e.g., [10,43,44]). Such indexes are not being used on transaction processing systems. Instead, they target business intelligence workloads, which lack updates and therefore do not need complicated concurrency control mechanisms. PLP eliminates the need for latching and concurrency control at the index level. Therefore, we expect to get a significant performance boost if we substitute the index implementation with a cache-friendlier B+tree alternative, since the B+tree probes are the most expensive remaining component of PLP.

7.3 Dynamic load balancing

There is a large body of related work, but most of it focuses on clustered (shared-nothing) environments. For example, [1] analyzes and compares different approaches for index reorganization during repartitioning in shared-nothing deployments. Lee et al. [30] propose an index structure similar to the *MRBtree*, which eases the index reorganization during repartitioning in a shared-nothing system and Mondal et al. [36] extend this design by keeping statistics for each branch pointed by the root node of a partition's subtree. While the structure of [36] enables the observation of access patterns at a fine granularity all the accesses have the same weight, no matter how recent or old they are. Our *two-level aging-based histogram* assigns higher weight to the recent accesses. This allows us to have a more accurate view of the skewed access patterns and detect load imbalances quickly.

Shinobi [55] uses a cost model to decide whether the benefits of a new partitioning configuration worth to pay the cost of repartitioning. Shinobi focuses on insert-heavy workloads where data is rarely queried and when queried the queries focus on a small region of the most recently inserted records. Its benefits primarily come from avoiding to index the large infrequently accessed parts of the database. We consider

mainstream transactional workloads where the entire database is accessed and we cannot drop any indexes.

The histogram-based technique we use is influenced from previous work on maintaining dynamic histograms on data distributions for accurately estimating the selectivity of query predicates [14,15]. In our case, we are interested in the frequency of accesses to a particular region, rather than the data distribution, and on the access pattern.

Finally, our work is orthogonal to techniques that decide initial partitioning configuration. Schism [11] creates partitions to minimize the number of distributed transactions by representing the workload as a graph and using a graph partitioning algorithm. While, in [45] the query optimizer is used to get suggestions for the initial partitions. These tools only create the initial configuration; if the workload characteristics change over time, however, the initial configuration is useless and the system has to re-calculate the partitioning configuration and perform the repartitioning.

8 Limitations of PLP

While we cannot find weaknesses in the MRBTree access method, PLP has some, most of them coming from its ancestor, data-oriented execution [40].

Applications that have less pressure on the storage manager. First of all, this system is designed for high performance transaction processing that imposes great pressure on the internal of the database storage layer. Thus, certain classes of applications may not benefit from it, or even get penalized. For example, for our evaluation we use the specialized `TATP` and `TPC-B` benchmarks instead of the more popular `TPC-C`. The reason for that is that our baseline systems (conventional and logically partitioned) does not encounter any of the issues we try to address in `TPC-C` and there is less room for improvement in most of the transactions, unlike `StockLevel` (Fig. 18).

Another example, are business intelligence applications with large file scans or joins. In such workloads PLP may penalize performance since it may require transferring large volumes of data among participating threads. It is common practice, however, to employ dedicated database engines (usually column-stores [50]) for such workloads.

Non-partition aligned index accesses. PLP partitions each table using range-based partitioning to the keys of a specific subset of the columns of the table. The DBA, however, may have decided to build indexes (usually non-clustered secondary indexes) that do not contain the columns that PLP uses for the partitioning. We refer to such indexes as *non-partitioning aligned indexes* and they may become performance bottlenecks. In data-oriented execution and PLP we handle such accesses by appending each index leaf entry with the fields of the record that are needed for identifying the partition-owning thread. The non-partitioning aligned index is accessed as a conventional index, without avoiding any locking or latching, in order to retrieve the id of the record to be accessed in the heap file and then the access is passed to the appropriate thread.

As Fig. 19 shows, such accesses can be burdensome for PLP. However, as a proactive measure, we implemented tools that help the application developer and the DBA to avoid having workloads with very frequent such index accesses [41].

Breaking the transactions. As mentioned previously (Sect. 3.1), the transactions need to be divided into smaller actions based on the data accessed in different parts of the transaction. These actions are represented as a directed graph to understand the transaction flow and dependencies among the actions. This representation also helps us to exploit intra-transaction parallelism for the independent actions. However, it introduces the initial cost of identifying these actions. We implemented a tool that automatically forms such a transaction flow graph given SQL statement [41] to ease this initial cost.

9 PLP on future hardware and conclusions

Unlike conventional systems, which either embrace fully shared-everything or shared-nothing philosophies, physiological partitioning takes the best features of both to produce a hybrid system that operates nearly latch and lock free, while still retaining the convenience of a common underlying storage pool and log. We achieve this result with a new multi-rooted B+tree structure and careful assignment of threads to data. This design and the MRBtree structure also allow us to have a lightweight, robust, and efficient dynamic load balancing and repartitioning mechanism.

As multicore hardware trends evolve, PLP becomes increasingly attractive for several reasons. Conventional OLTP is ill-suited to modern and upcoming hardware since; (a) The code of an OLTP system is full of unscalable critical sections [24], (b) The access patterns are unpredictable [48] that even the most advanced prefetchers fail to detect [47], (c) The majority of the accesses are shared read-write, and hence, they under-perform on caches with non-uniform access latency [4,17].

As we have seen, PLP, combined with previous advances in logging, eliminates all three problems. The majority of unscalable critical sections are completely eliminated, access patterns are regularized by the thread assignments, and threads no longer share data to communicate, eliminating the shared R/W problem.

This regularity will become increasingly important as hardware continues to make more and more demands of the software. For example, it is almost inevitable that processor cache access latencies will be non-uniform [4,17,18].

Unfortunately, OLTP will only be able to utilize effectively these new architectures if it can eliminate the majority of accesses that are shared among multiple processors.

In short, by eliminating a large class of communication, PLP leaves OLTP engines much better-poised to take advantage of upcoming hardware, whatever form it may take.

Acknowledgments The authors are deeply grateful for all the members of Data-Intensive Applications and Systems (DIAS) laboratory of École Polytechnique Fédérale de Lausanne who made this work possible through their research efforts, helpful feedback, and encouragement. We would also like to thank the anonymous reviewers whose many thoughtful and constructive remarks helped improve this paper. This research has been supported by a PhD research fellowship from IBM; grants and equipment from Intel and Sun; a Sloan research fellowships; an IBM faculty partnership award; NSF grants CCR-0205544, CCR-0509356, IIS-0133686, and IIS-0713409; an ESF EurYI award; and Swiss National Foundation funds.

References

1. Achyutuni, K.J., Omiecinski, E., Navathe, S.B.: Two techniques for on-line index modification in shared nothing parallel databases. In: SIGMOD, pp. 125–136 (1996)
2. Agrawal, R., Carey, M.J., Livny, M.: Concurrency control performance modeling: alternatives and implications. ACM TODS **12**(4), 609–654 (1987)
3. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: SIGFIDET, pp. 107–141 (1970)
4. Beckmann, B.M., Wood, D.A.: Managing wire delay in large chip-multiprocessor caches. In: MICRO, pp. 319–330 (2004)
5. Bender, M.A., Fineman, J.T., Gilbert, S., Kuszmaul, B.C.: Concurrent cache-oblivious B-trees. In: SPAA, pp. 228–237 (2005)
6. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM TODS **8**(4), 465–483 (1983)
7. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1987)
8. Brewer, E.A.: Towards robust distributed systems (abstract). In: PODC, pp. 7–7 (2000)
9. Chen, S.: FlashLogging: exploiting flash devices for synchronous logging performance. In: SIGMOD, pp. 73–86 (2009)
10. Chen, S., Gibbons, P.B., Mowry, T.C., Valentin, G.: Fractal prefetching B+-Trees: optimizing both cache and disk performance. In: SIGMOD, pp. 157–168 (2002)
11. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. PVLDB **3**, 48–57 (2010)
12. Das, S., Antony, S., Agrawal, D., El Abbadi, A.: Thread cooperation in multicore architectures for frequency counting over multiple data streams. PVLDB **2**, 217–228 (2009)
13. Dewitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.i., Rasmussen, R.: The Gamma database machine project. IEEE Trans. Knowl. Data Eng. TKDE **2**(1), 44–62 (1990)
14. Donjerkovic, D., Ioannidis, Y.E., Ramakrishnan, R.: Dynamic histograms: Capturing evolving data sets. In: ICDE, p. 86 (2000)
15. Gibbons, P.B., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. ACM TODS **27**, 261–298 (2002)
16. Graefe, G.: Sorting and indexing with partitioned B-trees. In: CIDR, pp. 1–13 (2003)
17. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A.: Reactive NUCA: near-optimal block placement and replication in distributed caches. In: ISCA, pp. 184–195 (2009)
18. Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N., Ailamaki, A., Falsafi, B.: Database servers on chip multiprocessors: limitations and opportunities. In: CIDR, pp. 79–87 (2007)
19. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992 (2008)
20. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: CIDR, pp. 132–141 (2007)
21. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. Computer **41**, 33–38 (2008)
22. Jaluta, I., Sippu, S., Soisalon-Soininen, E.: B-tree concurrency control and recovery in page-server database systems. ACM TODS **31**, 82–132 (2006)
23. Johnson, R., Pandis, I., Ailamaki, A.: Improving OLTP scalability using speculative lock inheritance. PVLDB **2**(1), 479–489 (2009)
24. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-MT: a scalable storage manager for the multicore era. In: EDBT, pp. 24–35 (2009)
25. Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M., Ailamaki, A.: Aether: a scalable approach to logging. PVLDB **3**, 681–692 (2010)
26. Jones, E., Abadi, D.J., Madden, S.: Low overhead concurrency control for partitioned main memory databases. In: SIGMOD, pp. 603–614 (2010)
27. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM TODS **6**(2), 213–226 (1981)
28. Lahiri, T., Srihari, V., Chan, W., MacNaughton, N., Chandrasekaran, S.: Cache fusion: Extending shared-disk clusters with shared caches. In: VLDB, pp. 683–686 (2001)
29. Larson, P.A., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. PVLDB **5**(4), 298–309 (2011)
30. Lee, M.L., Kitsuregawa, M., Ooi, B.C., Tan, K.L., Mondal, A.: Towards self-tuning data placement in parallel database systems. In: SIGMOD, pp. 225–236 (2000)
31. Lightstone, S., Surendra, M., Diao, Y., Parekh, S.S., Hellerstein, J.L., Rose, K., Storm, A.J., Garcia-Arellano, C.: Control theory: a foundational technique for self managing databases. In: ICDE Workshops, pp. 395–403 (2007)
32. Lomet, D., Anderson, R., Rengarajan, T.K., Spiro, P.: How the Rdb/VMS data sharing system became fast. Technical Report CRL-92-4, Dec (1992)
33. Mohan, C.: ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In: VLDB, pp. 392–405 (1990)
34. Mohan, C., Levine, F.: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In: SIGMOD, pp. 371–380 (1992)
35. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA, pp. 253–262 (2005)
36. Mondal, A., Kitsuregawa, M., Ooi, B.C., Tan, K.L.: R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases. In: GIS, pp. 28–33 (2001)
37. Muth, P., O'Neil, P., Pick, A., Weikum, G.: The LHAM log-structured history data access method. VLDB J. **8**, 199–221 (2000)
38. Neuvonen, S., Wolski, A., Manner, M., Raatikka, V.: Telecom Application Transaction Processing Benchmark (TATP). http://tatpbenchmark.sourceforge.net/ (2009)
39. Oracle: Oracle real application clusters. Available at http://www.orace.com/technetwork/database/clustering/overview

40. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. PVLDB **3**(1), 928–939 (2010)
41. Pandis, I., Tözün, P., Branco, M., Karampinas, D., Porobic, D., Johnson, R., Ailamaki, A.: A data-oriented transaction execution engine and supporting tools. In: SIGMOD, pp. 1237–1240 (2011)
42. Pandis, I., Tözün, P., Johnson, R., Ailamaki, A.: PLP: page latch-free shared-everything OLTP. PVLDB **4**(10), 610–621 (2011)
43. Rao, J., Ross, K.A.: Cache conscious indexing for decision-support in main memory. In: VLDB, pp. 78–89 (1999)
44. Rao, J., Ross, K.A.: Making B+-trees cache conscious in main memory. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pp. 475–486 (2000)
45. Rao, J., Zhang, C., Megiddo, N., Lohman, G.: Automating physical database design in a parallel database. In: SIGMOD, pp. 558–569 (2002)
46. Sewall, J., Chhugani, J., Kim, C., Satish, N., Dubey, P.: PALM: Parallel architecture-friendly latch-free modifications to b+trees on many-core processors. PVLDB **4**(11), 795–806 (2011)
47. Somogyi, S., Wenisch, T.F., Ailamaki, A., Falsafi, B.: Spatio-temporal memory streaming. In: ISCA, pp. 69–80 (2009)
48. Somogyi, S., Wenisch, T.F., Hardavellas, N., Kim, J., Ailamaki, A., Falsafi, B.: Memory coherence activity prediction in commercial workloads. In: WMPI, pp. 37–45 (2004)
49. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**, 4–9 (1986)
50. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-store: a column-oriented DBMS. In: VLDB, pp. 553–564 (2005)
51. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: VLDB, pp. 1150–1160 (2007)
52. Thomasian, A.: Concurrency control: methods, performance, and analysis. ACM Comput. Surv. 30, 70–119 (1998)
53. TPC: TPC benchmark B standard specification, revision 2.0 (1994). Available at http://www.tpc.org/tpcb
54. TPC: TPC benchmark C (OLTP) standard specification, revision 5.11 (2010). Available at http://www.tpc.org/tpcc
55. Wu, E., Madden, S.: Partitioning techniques for fine-grained indexing. In: ICDE, pp. 1127–1138 (2011)